Constraint-based Relational Verification

Hiroshi Unno[1,2], Tachio Terauchi[3], and Eric Koskinen[4]

[1] University of Tsukuba, Ibaraki, Japan
[2] RIKEN AIP, Tokyo, Japan
[3] Waseda University, Tokyo, Japan
[4] Stevens Institute of Technology, New Jersey, USA

Abstract. In recent years they have been numerous works that aim to automate relational verification. Meanwhile, although Constrained Horn Clauses (CHCs) empower a wide range of verification techniques and tools, they lack the ability to express hyperproperties beyond k-safety such as generalized non-interference and co-termination.
This paper describes a novel and fully automated constraint-based approach to relational verification. We first introduce a new class of predicate Constraint Satisfaction Problems called pfwCSP where constraints are represented as clauses modulo first-order theories over predicate variables of three kinds: ordinary, well-founded, or functional. This generalization over CHCs permits arbitrary (i.e., possibly non-Horn) clauses, well-foundedness constraints, functionality constraints, and is capable of expressing these relational verification problems. Our approach enables us to express and automatically verify problem instances that require non-trivial (i.e., non-sequential and non-lock-step) self-composition by automatically inferring appropriate *schedulers* (or *alignment*) that dictate when and which program copies move. To solve problems in this new language, we present a constraint solving method for pfwCSP based on *stratified* CounterExample-Guided Inductive Synthesis (CEGIS) of ordinary, well-founded, and functional predicates.
We have implemented the proposed framework and obtained promising results on diverse relational verification problems that are beyond the scope of the previous verification frameworks.

Keywords: relational verification, constraint solving, CEGIS

1 Introduction

We describe a novel constraint-based approach to automatically solving a wide range of relational verification problems including k-safety, co-termination [6, 10], termination-sensitive non-interference (TS-NI) [63], and generalized non-interference (GNI) [41] for infinite-state programs.

A key challenge in relational property verification is the discovery of *relational invariants* which relate the states of multiple program executions. However, whereas most prior approaches must fix the execution *schedule*[5] (e.g., lock-step or sequential) [8, 20, 21, 43, 55, 58], a recent work by Shemer et al. [51] has

[5] The notion of *schedule* is also often called an *alignment* in literature.

proposed a method to automatically infer sufficient *fair* schedulers to prove the goal relational property. Importantly, the schedulers in their approach can be *semantic* in which the choice of which program to execute can depend on the *states* of the programs as opposed to the classic *syntactic* schedulers such as lock-step and sequential that can only depend on the control locations. However, their approach requires the user to provide appropriate atomic predicates and is not fully automatic. Moreover, they only support k-safety properties. A recent work has proposed a method for automatically verifying non-hypersafety relational properties but only for *finite* state systems [19].

Meanwhile, today's constraint-based frameworks are also insufficient at automating relational verification. The class of predicate constraints called Constrained Horn Clauses (CHCs) [13] has been widely adopted as a "common intermediate language" for uniformly expressing verification problems for various programming paradigms, such as functional and object-oriented languages. Example uses of the CHCs framework include safety property verification [29,30,36] and refinement type inference [33,37,54,57,66]. The separation of constraint generation and solving has facilitated the rapid development of constraint generation tools such as RCaml [57], SeaHorn [30], and JayHorn [36] as well as efficient constraint solving tools such as SPACER [38], Eldarica [32], and HoIce [14]. Unfortunately, CHCs lack the ingredients to sufficiently express these relational verification problems.

In this paper we introduce automated support for relational verification by generalizing CHCs and introducing a new class of predicate Constraint Satisfaction Problems called pfwCSP. This language allows constraints that are *arbitrary (i.e., possibly non-Horn)* clauses modulo first-order theories over predicate variables that can be *functional predicates*, *well-founded predicates* or ordinary predicates. We then show that, thanks to the enhanced predicate variables, pfwCSP can express *non-hypersafety* relational properties such as co-termination [11], termination-sensitive non-interference (TS-NI) [63], and generalized non-interference (GNI) [41]. In addition, our approach effectively quantifies over the schedule, expressing *arbitrary fair semantic scheduling* thanks to non-Horn clauses and functional predicates (functional predicates are needed to express fairness in the presence of non-termination which is needed for properties like co-termination and TS-GNI). The flexibility allows our approach to automatically discover a fair semantic schedule and verify difficult relational problem instances that require non-trivial schedules. We prove that our encodings are *sound* and *complete*. Expressing relational invariants with such flexible scheduling is not possible with CHCs. However, pfwCSP retains a key benefit of CHCs: the idea of separating constraint generation from solving.

We next present a novel constraint solving method for pfwCSP based on *stratified* CounterExample-Guided Inductive Synthesis (CEGIS) of ordinary, well-founded, and functional predicates. In our method, ordinary predicates represent relational inductive invariants, well-founded predicates witness synchronous termination, and functional predicates represent Skolem functions witnessing existential quantifiers that encode angelic non-determinism. These witnesses for a

relational property are often mutually dependent and involve many variables in a complicated way (see Appendix B, D, and F for examples). The synthesis thus needs to use expressive templates without compromising the efficiency. Stratified CEGIS combines CEGIS [52] with stratified families of templates [56] (*i.e.*, decomposing templates into a series of increasingly expressive templates) to achieve completeness in the sense of [34,56], a theoretical guarantee of convergence, and a faster and stable convergence by avoiding the overfitting problem of expressive templates to counterexamples [45]. The constraint solving method naturally generalizes a number of previous techniques developed for CHCs solving and invariant/ranking function synthesis, addressing the challenges due to the generality of pfwCSP that is essential for relational verification.

We have implemented the above framework and have applied our tool PCSat to a diverse collection of 20 relational verification problems and obtained promising results. The benchmark problems go beyond the capabilities of the existing related tools (such as CHCs solvers and program verification tools). PCSat has solved 15 problems fully automatically by synthesizing complex witnesses for relational properties, and for the 5 problems that could not be solved fully automatically within the time limit, PCSat was able to solve them semi-automatically provided that a part of an invariant is manually given as a hint.

2 Overview

2.1 Relational verification problems

***k*-safety** Consider the following program taken from [51] that uses a summation to calculate the square of `x`, and then doubles it.

```
doubleSquare(bool h, int x) {
   int z, y=0;
   if (h) { z = 2*x; } else { z = x; }
   while (z>0) { z--; y = y+x; }
   if (!h) { y = 2*y; }
   return  y;
}
```

This program also takes another input `h` and, if the value of `h` is true, calculates the result differently. The classical relational property *termination-insensitive non-interference* (TI-NI) says that, roughly, an observer cannot infer the value of high security variables (`h` in this case) by observing the outputs (`y`). This is a *2-safety property* [17, 55]: it relates two executions of the same program. In this example, we ask whether two executions that initially agree on `x` (*i.e.*, $\mathtt{x}_1 = \mathtt{x}_2$) will agree on the resulting `y` (*i.e.*, $\mathtt{y}_1 = \mathtt{y}_2$). The subscripts in these relations indicate copies of the program: $\mathtt{x}_1$ is variable `x` in the first copy of the program and $\mathtt{x}_2$ is variable `x` in the second copy. More generally, k-safety means that if the initial states of a k-tuple of programs satisfy a pre-relation *Pre*, then when they all terminate the k-tuple of post states will satisfy post-relation *Post*.

The literature proposes many ways to reason about k-safety including methods of reducing a multi-program problem to a single-program problem, such as through self-composition [8, 55, 58], product programs [7], and their variants [21,47,51,53,59]. Their key challenge is that of *scheduling*: how to interleave the programs' executions so that invariants in the combined program are able to effectively describe cross-program relationships. Indeed, as proved by [51], verifying this example with the naïve lock-step scheduling is impossible with only linear arithmetic invariants while linear arithmetic invariants suffice with a more "semantic" scheduling that schedules the copy with $\mathsf{h}_1 = \mathtt{false}$ to iterate the loop twice per each iteration of the loop in the copy with $\mathsf{h}_2 = \mathtt{true}$.

In this paper, we will describe a way to pose the scheduling problem as a part of a series of constraints so that the search for an effective scheduler is relegated to the solver level. In our approach, a k-safety verification problem is encoded as a set of constraints containing (ordinary) predicate variables that represent the scheduler to be discovered and a relational invariant preserved by the scheduler. Specially, we introduce a predicate variable inv that represents a relational invariant and for each $A \subseteq \{1, \ldots, k\}$, a predicate variable $\mathsf{sch}_A(\widetilde{V}_1, \ldots, \widetilde{V}_k)$ where $\widetilde{V}_i$ are the variables of the ith program, and add constraints that say that if the predicate is $\mathtt{true}$, then the programs whose index are in A will step forward while the rest remain still and also inv is preserved by the step. For soundness, it is important to constrain the scheduler to be *fair*, *i.e.*, at least one program that can progress must be scheduled to progress if there is a program that can progress. As we shall show in Sec. 4, non-Horn clauses are essential to expressing the fairness constraint. Roughly, the idea is to use a clause with multiple positive predicate variables (*i.e.*, *head disjunction*) to say "*if the relational invariant holds, then at least one of the unfinished programs must be scheduled to progress.*"

Our approach is similar to and is inspired by the approach of [51] that also infers a fair semantic scheduler. However, their approach requires the user to provide sufficient atomic predicates manually and is not fully automated. By contrast, our approach soundly-and-completely encodes the k-safety verification problem together with scheduler inference as a set of constraints thanks to the expressiveness of pfwCSP, and automatically solves those constraints by the stratified CEGIS algorithm (cf. Sec. 7 for further comparison).

Co-termination Now consider the following pair of programs.

P_1^{cot} : `while (x>0) { x = x - y; }`
P_2^{cot} : `while (x>0) { x = x - 2 × y; }`

A (non-safety) relational question is whether these programs P_1^{cot} and P_2^{cot} agree on termination [6, 10]. In general they do not: if, for example, P_1^{cot} is executed with $\mathsf{x} < 0$ and P_2^{cot} with $\mathsf{x} > 0 \wedge \mathsf{y} = 0$, the first will terminate while the second will diverge. However, under the pre-relation $Pre \equiv \mathsf{x}_1 = \mathsf{x}_2 \wedge \mathsf{y}_1 = \mathsf{y}_2$, they will *agree* on termination: the first program terminates iff the second one does. The property falls outside of the k-safety fragment as it cannot be refuted by finite execution traces. It is worth noting that *termination-sensitive non-interference*

(TS-NI) is the conjunction of TI-NI and co-termination of two copies of the same target program with *Pre* equating the copies' low inputs.

Proving co-termination, like k-safety, can be aided by scheduler and we can again use our constraints over predicate variables. But this is not enough. We need additional constraints to ensure that whenever one of the two has terminated, the other is also guaranteed to terminate. To address this, we next introduce *well-founded predicate variables*. These predicate variables will appear in our generalized language of constraints as terms of the form $\mathsf{wfr}(\widetilde{V}_i, \widetilde{V}_i')$, where the relation wfr must be *discovered* by the constraint solving method. (In Sec. 5 we describe how to achieve this through our stratified CEGIS algorithm.) For the above example, our stratified CEGIS algorithm and our tool PCSat automatically discovers (1) a schedule where the two programs step together when $x_1 > 0$ and $x_2 > 0$, (2) a relational invariant that implies that if the first program is terminated, then either the second program is terminated or $y_2 \geq 1$ (and vice-versa), and (3) well-founded relations that (combined with the relational invariant) witness that if the loop has terminated in the second program ($x_2 \leq 0$) but not in the first ($x_1 > 0$), then a transition in the first is well-founded (and vice-versa). In Sec. 4, we show how co-termination problems can be soundly-and-completely encoded in pfwCSP.

Generalized non-interference. Now consider the following program.

```
gniEx(bool high, int low) {
  if (high) {
    int x = *^int; if (x >= low) { return x; } else { while (true) {} }
  } else {
    int x = low; while (*^bool) { x++; } return x;
  }
}
```

The $*^{\texttt{int}}$ (resp. $*^{\texttt{bool}}$) above indicates an integer (resp. a binary) non-deterministic choice. *Termination-insensitive generalized non-interference* (TI-GNI) [41] is an extension of non-interference to non-deterministic programs, and it says that for any two copies of the program with possibly different values for the high security input (`high` in this example) and with the same value for the low security input (`low` in this example), if one copy has a terminating execution that ends in some output (the final value of `x` in this example), then the other copy has either a terminating execution ending in the same output or a non-terminating execution. The *termination-sensitive* variant (TS-GNI) strengthens the condition by asserting that if one copy has a terminating execution then the other copy has a terminating execution that ends in the same output. Both GNI variants are $\forall\exists$ hyperproperties and fall outside of the k-safety fragment.

Verifying GNI requires handling non-determinism. Note that non-determinism occurs both *demonically* (*i.e.*, as $\forall$) and *angelically* (*i.e.*, as $\exists$) in GNI. While handling demonic non-determinism is straightforward in a constraint-based verification since the term variables are implicitly universally quantified, handling angelic non-determinism is less straightforward.

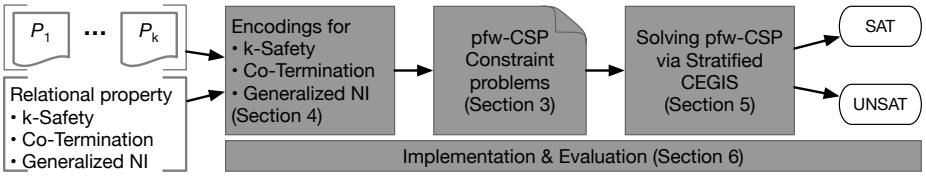

Fig. 1. Overview of the contributions and how they achieve a constraint-based strategy for relational verification.

Our approach handles finitary angelic non-determinism like $*^{\texttt{bool}}$ by adding non-Horn clauses with head disjunctions that roughly express the condition "*the relational invariant remains true in one of the finitely many next step choices*". To handle infinitary non-determinism like $*^{\texttt{int}}$, we introduce *functional predicate variables* denoted $\mathsf{f}(\widetilde{V}, r)$. In these terms, f is a predicate variable to be discovered but with a new wrinkle: this predicate involves a return value r and the interpretation of f is a *total function* over $\widetilde{V}$. For this example, we introduce the term $\mathsf{f}(\widetilde{V}, r)$ where r represents the value chosen non-deterministically at $*^{\texttt{int}}$ and $\widetilde{V}$ are program variables and *prophecy variables* that represent the final return values of the demonic copy. For this example, PCSat automatically discovers the predicate $r = \texttt{ret}_1$ where $\texttt{ret}_1$ is the prophecy variable for the return value of the demonic copy. With it, PCSat is able to verify TS-GNI and TI-GNI of this example. We remark that functional predicates are also used to encode scheduler fairness in the presence of non-termination and is needed to ensure soundness for properties like co-termination and TS-GNI. In Sec. 4.3, we show how TI-GNI and TS-GNI can be soundly-and-completely encoded in pfwCSP.

2.2 Challenges & Contributions

There are several challenges that we face in supporting relational verification problems with a constraint-based approach. The subsequent sections of this paper are organized around addressing those challenges as follows:

- We first ask how to generalize the constraint language to go beyond CHCs to express a more general class of relational verification problems. To this end, in Sec. 3, we present a new language called *predicated constraint satisfaction problems* (pfwCSP), which incorporate non-Horn clauses, (ordinary) predicate variables, well-founded predicate variables, and functional predicate variables.
- We next return to the above relational verification problems –k-safety, co-termination, and generalized non-interference– and describe how pfwCSP can express each of them in a sound and complete manner in Sec. 4.
- The next major contribution of our research is a novel *stratified* CEGIS algorithm for solving pfwCSP constraints. Our approach integrates advanced verification techniques: *stratified family of templates* [56] and *CEGIS of invariants/ranking functions* [14, 26, 28, 46]. While the individual ideas have

been proposed previously, they have only been designed for less expressive frameworks such as CHCs, and substantial extensions are needed to combine and apply them to the new pfwCSP framework as we shall show in Sec. 5.
– We next turn to an implementation and experimental validation on a diverse collection of 20 relational verification problems, consisting of k-safety problems from Shemer *et al.* [51] and new co-termination and GNI problems in Sec. 6. As far as we know, none of the existing *automated* tools other than our new tool called PCSat can solve them.

In sum, Fig. 1 depicts each of these sections and how, together, they enable relational verification. For space, the proofs of the soundness and completeness theorems are deferred to the appendix.

3 Predicate Constraint Satisfaction Problems pfwCSP

As discussed in Sec. 2, CHCs are insufficient to express important relational verification problems. In the section we introduced a generalized language of constraints called pfwCSP. The language of constraint satisfaction problems (CSP) permits non-Horn clauses, **p**redicate variable terms, including those for **f**unctional predicates and **w**ell-founded relations (pfw). We now define pfwCSP.

Let $\mathcal{T}$ be a (possibly many-sorted) first-order theory with the signature Σ. The syntax of $\mathcal{T}$-formulas and $\mathcal{T}$-terms is:

$$\begin{aligned}\text{(formulas) } \phi &::= X(\widetilde{t}) \mid p(\widetilde{t}) \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2\\ \text{(terms) } t &::= x \mid f(\widetilde{t})\end{aligned}$$

Here, the meta-variables x and X respectively range over term and predicate variables. The meta-variables p and f respectively denote predicate and function symbols of Σ. We use s as a meta-variable ranging over sorts of the signature Σ. We write $\star$ for the sort of propositions and $s_1 \to s_2$ for the sort of functions from s_1 to s_2. We write $\mathrm{ar}(o)$ and $\mathrm{sort}(o)$ respectively for the arity and the sort of a syntactic element o. A function f represents a constant if $\mathrm{ar}(f) = 0$. We write $\mathit{ftv}(\phi)$ and $\mathit{fpv}(\phi)$ respectively for the set of free term and predicate variables that occur in ϕ. We write $\widetilde{x}$ for a sequence of term variables, $|\widetilde{x}|$ for the length of $\widetilde{x}$, and ϵ for the empty sequence. We often abbreviate $\neg\phi_1 \vee \phi_2$ as $\phi_1 \Rightarrow \phi_2$. We henceforth consider only well-sorted formulas and terms. We use φ as a meta-variable ranging over $\mathcal{T}$-formulas without predicate variables.

We now define a pCSP $\mathcal{C}$ (with ordinary but without well-founded and functional predicate variables) to be a finite set of clauses of the form

$$\varphi \vee \left(\bigvee_{i=1}^{\ell} X_i(\widetilde{t_i})\right) \vee \left(\bigvee_{i=\ell+1}^{m} \neg X_i(\widetilde{t_i})\right) \tag{1}$$

where $0 \le \ell \le m$. We write $\mathit{ftv}(c)$ for the set of free term variables of a clause c. The set of free term variables of $\mathcal{C}$ is defined by $\mathit{ftv}(\mathcal{C}) = \bigcup_{c\in\mathcal{C}} \mathit{ftv}(c)$. We regard the variables in $\mathit{ftv}(c)$ as implicitly universally quantified. We write $\mathit{fpv}(\mathcal{C})$

for the set of free predicate variables that occur in $\mathcal{C}$. A *predicate substitution* σ is a finite map from predicate variables X to closed predicates of the form $\lambda x_1, \ldots, x_{\mathrm{ar}(X)}.\varphi$. We write $\sigma(\mathcal{C})$ for the application of σ to $\mathcal{C}$ and $\mathrm{dom}(\sigma)$ for the domain of σ. We call σ a *syntactic solution* for $\mathcal{C}$ if $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\sigma)$ and $\models \bigwedge \sigma(\mathcal{C})$. Similarly, we call a predicate interpretation ρ a *semantic solution* for $\mathcal{C}$ if $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\rho)$ and $\rho \models \bigwedge \mathcal{C}$.

Remark 1. The language pCSP generalizes over existing languages of constraints. CHCs can be obtained as a restriction of pCSP where $\ell \leq 1$ in (1) for all clauses. We can also define coCHCs as pCSP but with the restriction that $m \leq \ell + 1$ for all clauses. A linear CHCs is a pCSP that is both CHCs and coCHCs.

We next extend pCSP to pfwCSP by adding well-foundedness and functionness constraints. A pfwCSP $(\mathcal{C}, \mathcal{K})$ consists of

- a finite set $\mathcal{C}$ of pCSP-clauses over predicate variables and
- a kinding function $\mathcal{K}$ that maps each predicate variable $X \in \mathit{fpv}(\mathcal{C})$ to its kind: any one of $\bullet$, $\Downarrow$, or λ which respectively represent ordinary, well-founded, and functional predicate variables.

We write $\rho \models \mathit{WF}(X)$ if the interpretation $\rho(X)$ of the predicate variable X is *well-founded*, that is, $\mathrm{sort}(X) = (\widetilde{s}, \widetilde{s}) \to \star$ for some $\widetilde{s}$ and there is no infinite sequence $\widetilde{v}_1, \widetilde{v}_2, \ldots$ of sequences $\widetilde{v}_i$ of values of the sorts $\widetilde{s}$ such that $(\widetilde{v}_i, \widetilde{v}_{i+1}) \in \rho(X)$ for all $i \geq 1$. We write $\rho \models \mathit{FN}(X)$ if X is *functional*, that is, $\mathrm{sort}(X) = (\widetilde{s}, s) \to \star$ for some $\widetilde{s}$ and s, and $\rho \models \forall \widetilde{x} \colon \widetilde{s}.(\exists y \colon s.X(\widetilde{x}, y)) \wedge \forall y_1, y_2 \colon s.(X(\widetilde{x}, y_1) \wedge X(\widetilde{x}, y_2) \Rightarrow y_1 = y_2)$ holds. We call a predicate interpretation ρ a *semantic solution* for $(\mathcal{C}, \mathcal{K})$ if ρ is a semantic solution of $\mathcal{C}$, $\rho \models \mathit{WF}(X)$ for all X such that $\mathcal{K}(X) = \Downarrow$, and $\rho \models \mathit{FN}(X)$ for all X such that $\mathcal{K}(X) = \lambda$. The notion of syntactic solution can be similarly generalized to pfwCSP.

Definition 1 (Satisfiability of pfwCSP**).** The predicate satisfiability problem of a pfwCSP $(\mathcal{C}, \mathcal{K})$ is that of deciding whether it has a semantic solution.

Remark 2. Recall that we assume that the $\mathcal{T}$-formulas φ in pCSP clauses do not contain quantifiers. The assumption, however, is not a restriction for pfwCSP because we can Skolemize quantifiers using functional predicates.

4 Relational Verification with Constraints

We now present reductions from relational verification problems to pfwCSP, thus enabling a new route to automation of these problems. We begin with k-safety, and then move toward liveness and non-determinism, which are thorny problems in the relational setting. We first provide some basic definitions and notations.

Programs. We consider programs $P_1,\ldots,P_k$ on variables $\widetilde{V_1},\ldots,\widetilde{V_k}$, respectively. A *state* of the program P_i is a valuation of the variables $\widetilde{V_i}$. We represent such a valuation by a sequence of values $\widetilde{v}$ such that $|\widetilde{v}| = |\widetilde{V_i}|$. We assume that each P_i is defined by the predicate $T_i(\widetilde{V_i}, \widetilde{V_i}')$ denoting its one-step transition relation i.e., $T_i(\widetilde{v}, \widetilde{v}')$ implies that evaluating P_i one step from the state $\widetilde{v}$ reaches the state $\widetilde{v}'$. We also assume that there is a predicate $F_i(\widetilde{V_i})$ that represents the final states of the program such that $F_i(\widetilde{v})$ and $T_i(\widetilde{v}, \widetilde{v}')$ implies $\widetilde{v} = \widetilde{v}'$, i.e., the program self-loops when it reaches a final state. We say that a state $\widetilde{v}$ (multi-step) reaches a final state $\widetilde{v}'$ in the evaluation of P_i, written $\widetilde{v} \rightsquigarrow_i \widetilde{v}'$, if there exists a non-empty finite sequence of states π such that $\pi[1] = \widetilde{v}$, $\pi[|\pi|] = \widetilde{v}'$, $T_i(\pi[j-1], \pi[j])$ for all $1 < j \leq |\pi|$, and $F_i(\widetilde{v}')$. We write $\widetilde{v} \rightsquigarrow_i \bot$ if there exists a non-terminating evaluation from $\widetilde{v}$ in P_i, *i.e.*, if there exists an infinite sequence of states ϖ such that $\varpi[1] = \widetilde{v}$, $T_i(\varpi[j-1], \varpi[j])$ for all $1 < j$, and $\neg F_i(\varpi[j])$ for all $0 < j$. We note that a program may be non-deterministic, that is, $T_i(\widetilde{v}, \widetilde{v}')$ and $T_i(\widetilde{v}, \widetilde{v}'')$ may both be true for some $\widetilde{v}' \neq \widetilde{v}''$.

4.1 *k*-Safety

A *k-safety property* is given by predicates $Pre(\widetilde{V})$ and $Post(\widetilde{V})$ that respectively denote the pre and the post relations across the k-tuple.

Definition 2 (k-safety). The *k-safety property verification problem* is to decide if the following holds:

$$\forall \widetilde{v} = \widetilde{v_1},\ldots,\widetilde{v_k}.\forall \widetilde{v}' = \widetilde{v_1}',\ldots,\widetilde{v_k}'.Pre(\widetilde{v}) \wedge \textstyle\bigwedge_{i\in[k]} \widetilde{v_i} \rightsquigarrow_i \widetilde{v_i}' \Rightarrow Post(\widetilde{v}')$$

That is, any k-tuple of final states reachable from a k-tuple of states satisfying the pre-condition satisfies the post-condition. For instance, the TI-NI verification from Sec. 2.1 is a 2-safety property where P_1 and P_2 are copies of the same program, *Pre* states that the low inputs of the two programs are equal (*i.e.*, $\mathsf{x}_1 = \mathsf{x}_2$ in the example), and *Post* states that the low outputs of the two programs are equal (*i.e.*, $\mathsf{y}_1 = \mathsf{y}_2$ in the example).

We now describe a new way to pose the k-safety relational verification problem via constraints written in pfwCSP. We write $[k]$ for the set $\{1,\ldots,k\}$. We define $\mathcal{P}^+[k] = \{S \subseteq [k] \mid S \neq \emptyset\}$. Let $\widetilde{V} = \widetilde{V_1},\ldots,\widetilde{V_k}$ be a k-tuple of vectors, corresponding to the variables of the k programs.

Definition 3 (k-safety through constraints). We define pfwCSP constraints $\mathcal{C}_\mathrm{S}$ be the set of following clauses:

(1) $Pre(\widetilde{V}) \Rightarrow \mathsf{inv}(\widetilde{V})$
(2) $\mathsf{inv}(\widetilde{V}) \wedge \bigwedge_{i\in[k]} F_i(\widetilde{V_i}) \Rightarrow Post(\widetilde{V})$
(3) For each $A \in \mathcal{P}^+[k]$,
$\mathsf{inv}(\widetilde{V}) \wedge \mathsf{sch}_A(\widetilde{V}) \wedge \bigwedge_{i\in A} T_i(\widetilde{V_i}, \widetilde{V_i}') \wedge \bigwedge_{i\in[k]\setminus A} \widetilde{V_i} = \widetilde{V_i}' \Rightarrow \mathsf{inv}(\widetilde{V}')$
(4) For each $A \in \mathcal{P}^+[k]$, $\mathsf{inv}(\widetilde{V}) \wedge \mathsf{sch}_A(\widetilde{V}) \wedge \bigvee_{i\in[k]} \neg F_i(\widetilde{V_i}) \Rightarrow \bigvee_{i\in A} \neg F_i(\widetilde{V_i})$
(5) $\mathsf{inv}(\widetilde{V}) \wedge \bigvee_{i\in[k]} \neg F_i(\widetilde{V_i}) \Rightarrow \bigvee_{A\in\mathcal{P}^+[k]} \mathsf{sch}_A(\widetilde{V})$.

Here, inv and sch_A (for each $A \in \mathcal{P}^+[k]$) are ordinary predicate variables. Roughly, the predicate variables sch_A describe a *scheduler*. The scheduler stipulates that when $\mathsf{sch}_A(\widetilde{v_1}, \ldots, \widetilde{v_k})$ is true, each P_i such that $i \in A$ takes a step from the state $\widetilde{v_i}$ while the others remain still. Note that the scheduler is *semantic* in the sense that which programs are scheduled to be executed next can depend on the current states of the programs. Clauses (1)-(3) assert that inv is an invariant sufficient to prove the given safety property with the scheduler defined by sch_A's. Clauses (4) say that if an inv-satisfying state is such that the processes in A are allowed to move and some program has not yet terminated, then at least one process in A has not yet terminated. Clause (5) says that any state satisfying inv has to satisfy some sch_A. Clauses (4) and (5) ensure the *fairness* of the scheduler, that is, at least one unfinished program is scheduled to make progress if there is an unfinished program.

Theorem 1 (Soundness and Completeness of $\mathcal{C}_\mathrm{S}$). *The given k-tuple of programs satisfies the given k-safety property iff $\mathcal{C}_\mathrm{S}$ is satisfiable.*

We note that the soundness direction crucially relies on scheduler fairness. The completeness is with respect to semantic solutions (cf. Def. 1) and it is only "relative" with respect to syntactic solutions: a syntactic solution only exists when the predicates of the background theory are able to express sufficient invariants and schedulers (impossible in general for any decidable theory when the class of programs is Turing-powerful as in our case when the background theory of predicates is QFLIA).

It is important to note that $\mathcal{C}_\mathrm{S}$ is *not* CHCs because clause (5) has a head disjunction. $\mathcal{C}_\mathrm{S}$ may be seen as a constraint-based formulation of the approach proposed in [51]. However, their approach requires the user to provide sufficient predicates manually and is not fully automated, while our approach can fully automatically solve the problems by constraint solving (cf. Sec. 5).

Example 1. The formalization allows flexible scheduling. For instance, for the TI-NI example from Sec. 2.1, our approach is able to infer the predicate substitution that maps $\mathsf{sch}_{\{1\}}$, $\mathsf{sch}_{\{2\}}$, and $\mathsf{sch}_{\{1,2\}}$ to $\lambda\widetilde{V}.\mathsf{h}_1 \wedge \neg\mathsf{h}_2 \wedge \mathsf{z}_1 = 2\mathsf{z}_2$, $\lambda\widetilde{V}.\neg\mathsf{h}_1 \wedge \mathsf{h}_2 \wedge \mathsf{z}_2 = 2\mathsf{z}_1$, and $\lambda\widetilde{V}.(\mathsf{h}_1 \wedge \neg\mathsf{h}_2 \Rightarrow \mathsf{z}_1 + 1 = 2\mathsf{z}_2) \wedge (\neg\mathsf{h}_1 \wedge \mathsf{h}_2 \wedge \mathsf{z}_2 + 1 = 2\mathsf{z}_1)$ respectively, where $\widetilde{V}$ is the list of the variables in the two program copies. The inferred predicates stipulate that the copy with $\mathsf{h} = \mathtt{true}$ is scheduled to execute the loop two times per every loop iteration of the copy with $\mathsf{h} = \mathtt{false}$. Appendix A shows the pfwCSP encoding of the example. A solution generated by PCSat is also shown in Appendix B.

4.2 Co-termination

Intuitively, co-termination means that if one program terminates, then a second program must terminate [6,10]. This can also be thought of as a form of relational *termination problem*.[6]

[6] The property has also been called *relative termination* [31].

Definition 4 (Co-Termination). The *co-termination verification problem* is to decide if for all $\widetilde{v_1}, \widetilde{v_2}$ such that $Pre(\widetilde{v_1}, \widetilde{v_2})$, if $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1'}$ then $\widetilde{v_2} \not\rightsquigarrow_2 \bot$.

Roughly, the property says that from any pair of states related by *Pre*, if P_1 terminates, then P_2 must also terminate. Note that this is an *asymmetric* property. A symmetric version can be obtained by also asserting the property with the positions of the two programs exchanged. The symmetric version implies, assuming that there is at least one execution from any *Pre*-related state, that from any pair of *Pre*-related states, all executions from one state terminate iff all executions from the other one do as well. We now present an encoding of conditional co-termination in pfwCSP.

Definition 5 (Co-termination through constraints). Let $\widetilde{V} = \widetilde{V_1}, \widetilde{V_2}$. We define pfwCSP constraints $\mathcal{C}_{\mathrm{CoT}}$ be the set of following clauses:

(1) $Pre(\widetilde{V}) \wedge \mathsf{fnb}(\widetilde{V}, b) \Rightarrow \mathsf{inv}(0, b, \widetilde{V})$
(2) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \neg F_1(\widetilde{V_1}) \wedge \neg F_2(\widetilde{V_2}) \Rightarrow (-b \leq d \wedge d \leq b \wedge b \geq 0)$
(3a) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \mathsf{sch}_{\mathsf{FT}}(d, b, \widetilde{V}) \wedge T_2(\widetilde{V_2}, \widetilde{V_2}') \wedge (F_1(\widetilde{V_1}) \vee F_2(\widetilde{V_2}) \vee d' = d - 1) \Rightarrow \mathsf{inv}(d', b, \widetilde{V_1}, \widetilde{V_2}')$
(3b) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \mathsf{sch}_{\mathsf{TF}}(d, b, \widetilde{V}) \wedge T_1(\widetilde{V_1}, \widetilde{V_1}') \wedge (F_1(\widetilde{V_1}) \vee F_2(\widetilde{V_2}) \vee d' = d + 1) \Rightarrow \mathsf{inv}(d', b, \widetilde{V_1}', \widetilde{V_2})$
(3c) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \mathsf{sch}_{\mathsf{TT}}(d, b, \widetilde{V}) \wedge T_1(\widetilde{V_1}, \widetilde{V_1}') \wedge T_2(\widetilde{V_2}, \widetilde{V_2}') \Rightarrow \mathsf{inv}(d, b, \widetilde{V_1}', \widetilde{V_2}')$
(4a) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \mathsf{sch}_{\mathsf{FT}}(d, b, \widetilde{V}) \wedge \neg F_1(\widetilde{V_1}) \Rightarrow \neg F_2(\widetilde{V_2})$
(4b) $\mathsf{inv}(d, b, \widetilde{V}) \wedge \mathsf{sch}_{\mathsf{TF}}(d, b, \widetilde{V}) \wedge \neg F_2(\widetilde{V_2}) \Rightarrow \neg F_1(\widetilde{V_1})$
(5) $\mathsf{inv}(d, b, \widetilde{V}) \wedge (\neg F_1(\widetilde{V_1}) \vee \neg F_2(\widetilde{V_2})) \Rightarrow \bigvee_{a \in \{\mathsf{TT}, \mathsf{FT}, \mathsf{TF}\}} \mathsf{sch}_a(d, b, \widetilde{V})$
(6) $\mathsf{inv}(d, b, \widetilde{V}) \wedge F_1(\widetilde{V_1}) \wedge \neg F_2(\widetilde{V_2}) \wedge T_2(\widetilde{V_2}, \widetilde{V_2}') \Rightarrow \mathsf{wfr}(\widetilde{V_2}, \widetilde{V_2}')$

Here, $\mathsf{sch}_{\mathsf{TT}}$, $\mathsf{sch}_{\mathsf{FT}}$, and $\mathsf{sch}_{\mathsf{TF}}$ are 2-specialization of the k-safety scheduler of Def. 3. Clauses (3x)'s are similar to (3) of Def. 3 and assert that inv is an invariant under the scheduler. Clauses (4x)'s and (5), like (4) and (5) of Def. 3, are used to ensure the scheduler fairness. However, they are insufficient for co-termination as a non-terminating copy can be scheduled indefinitely leaving the other copy unscheduled. Clauses (1) and (2) are added to amend the issue. In (1), fnb is a functional predicate variable that is used to select a *bound* b, and (2) asserts that the *difference* d between the numbers of steps taken by the two copies is within b in any state in inv when neither copy has terminated. Note that d is initialized to 0 by (1) and properly updated in (3x)'s. Finally, by using the well-founded predicate variable wfr, (6) asserts that if P_1 has terminated, then so must eventually P_2.

Theorem 2 (Soundness and Completeness of $\mathcal{C}_{\mathrm{CoT}}$). *The given pair of programs co-terminate iff $\mathcal{C}_{\mathrm{CoT}}$ is satisfiable.*

As with Theorem 1, the soundness direction relies on scheduler fairness.

Example 2. Via the encoding, our PCSAT tool is able to verify the symmetric co-termination example from Sec. 2.1 by automatically inferring the solution described there. For space, the concrete constraint set and solution are given in Appendix C and D.

4.3 Generalized Non-Interference

We now turn to another relational property that cannot simply be captured by k-safety or co-termination. So-called *termination-insensitive (resp. -sensitive) generalized non-interference* (resp. TI-GNI, TS-GNI) are $\forall\exists$ hyperproperties: from any pre-related pair of states whenever one side can take a move to a post state, there must be a way for the other side to also move to a post state such that the post-relation holds. As remarked in Sec. 2, verifying GNI requires reasoning about both *demonic* (*i.e.*, for all) and *angelic* (*i.e.*, exists) *non-determinism*.

Definition 6 (TI/TS-GNI). The *GNI verification problem* is to decide if the following holds. If $\mathit{Pre}(\widetilde{v_1}, \widetilde{v_2})$ and $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$ then **(TI-GNI)** $(\exists \widetilde{v_2}'.\widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}' \wedge \mathit{Post}(\widetilde{v_1}', \widetilde{v_2}')) \vee \widetilde{v_2} \rightsquigarrow_2 \bot$; or **(TS-GNI)** $\exists \widetilde{v_2}'.\widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}' \wedge \mathit{Post}(\widetilde{v_1}', \widetilde{v_2}')$.

Note that our definition is parameterized by *Pre* and *Post*. The standard GNI definitions [41] can be obtained by letting P_1 and P_2 be copies of the same target program and letting *Pre* be the predicate equating the low inputs of the copies and *Post* be the predicate equating the low outputs of the copies.

To formalize the pfwCSP encodings of the GNI verification problems, we define a relation U_2 to be one such that $T_2(\widetilde{v}, \widetilde{v}') \Leftrightarrow \exists r.U_2(r, \widetilde{v}, \widetilde{v}')$ and $U_2(r, \widetilde{v}, \widetilde{v}') \wedge U_2(r, \widetilde{v}, \widetilde{v}'') \Rightarrow \widetilde{v}' = \widetilde{v}''$. Roughly, U_2 is a function version of the transition relation T_2 with the extra parameter r to make the non-deterministic choices explicit.

We now show the pfwCSP encodings of TI-GNI and TS-GNI. The key idea is to augment the encodings for k-safety and/or co-termination with *functional predicate variables* and *prophecy variables* that respectively represent the non-deterministic choices of the angelic side (*i.e.*, P_2) and the final outputs of the demonic side (*i.e.*, P_1).

Definition 7 (TI-GNI through constraints). We define pfwCSP constraints $\mathcal{C}_{\text{TIGNI}}$ as $\mathcal{C}_{\text{S}}$ in Def. 3 for $k = 2$ but with the following modifications:

(m1) The parameters representing the inputs and outputs of P_1 are extended with prophecy variables $\widetilde{p}$ where $|\widetilde{p}| = |\widetilde{V_1}|$. Accordingly, each occurrence of $\widetilde{V_1}$ is replaced by $\widetilde{p}, \widetilde{V_1}$, and each occurrence of $\widetilde{V_1}'$ is replaced by $\widetilde{p}', \widetilde{V_1}'$.
(m2) *Pre* is replaced by Pre' which is defined by $\mathit{Pre}'(\widetilde{p}, \widetilde{V_1}, \widetilde{V_2}) \Leftrightarrow \mathit{Pre}(\widetilde{V_1}, \widetilde{V_2})$, *i.e.*, the prophecy values are unconstrained in the pre-condition.
(m3) F_1 is replaced by F_1' defined by $F_1'(\widetilde{p}, \widetilde{V_1}) \Leftrightarrow F_1(\widetilde{V_1})$.
(m4) T_1 is replaced by T_1' defined by $T_1'(\widetilde{p}, \widetilde{V_1}, \widetilde{p}', \widetilde{V_1}') \Leftrightarrow T_1(\widetilde{V_1}, \widetilde{V_1}') \wedge \widetilde{p} = \widetilde{p}'$.
(m5) *Post* is replaced by Post' defined by $\mathit{Post}'(\widetilde{p}, \widetilde{V_1}, \widetilde{V_2}) \Leftrightarrow (\widetilde{p} = \widetilde{V_1} \Rightarrow \mathit{Post}(\widetilde{V_1}, \widetilde{V_2}))$, *i.e.*, if the prophecy was correct then the original post-condition must hold.
(m6) Each occurrence of $T_2(\widetilde{V_2}, \widetilde{V_2}')$ is replaced by $\mathsf{fnr}(\widetilde{p}, \widetilde{V_2}, r) \wedge U_2(r, \widetilde{V_2}, \widetilde{V_2}')$ where fnr is a functional predicate variable.

Modifications (m1)-(m5) concern prophecy variables. They are initialized arbitrarily as shown in (m2), propagated unmodified through the transitions as shown in (m4), and finally checked if they match P_1's outputs in (m5). Modification (m6) adds functional predicate variables to express the angelic non-deterministic choices of P_2. The functional predicate variables shift the onus of

making the right choices to the solver's task of discovering sufficient assignments to them. Importantly, the functional predicate takes the prophecy variables as parameters, thus allowing dependence on the final outputs of the demonic side.

Definition 8 (TS-GNI through constraints). We define pfwCSP constraints $\mathcal{C}_{\text{TSGNI}}$ as $\mathcal{C}_{\text{CoT}}$ in Def. 5 but with modifications of Def. 7 except (m3) and (m5), and with the following modifications:

(m3') F_1 is replaced by F_1' defined by $F_1'(\widetilde{p}, \widetilde{V_1}) \Leftrightarrow F_1(\widetilde{V_1}) \wedge \widetilde{p} = \widetilde{V_1}$.

(m5') The clause $\mathsf{inv}(\widetilde{p}, \widetilde{V_1}, \widetilde{V_2}) \wedge F_1'(\widetilde{p}, \widetilde{V_1}) \wedge F_2(\widetilde{V_2}) \Rightarrow \mathit{Post}(\widetilde{V_1}, \widetilde{V_2})$ is added.

$\mathcal{C}_{\text{TSGNI}}$ is similar to $\mathcal{C}_{\text{TIGNI}}$ except that it contains the difference bound and well-foundedness constraints to handle the "co-termination" aspect of TS-GNI, *i.e.*, if P_1 terminates and makes an output then P_2 must also be able to terminate and make a matching output. One subtle aspect of the encoding is that (m3') modifies the final state predicate for P_1 to enforce co-termination only when the prophecy is correct. However, it is worth noting that TS-GNI is *not* a conjunction of TI-GNI and co-termination. For instance, the GNI example from Sec. 2.1 satisfies TS-GNI but does not satisfy co-termination.

Theorem 3 (Soundess and Completeness of of TI-GNI). *The given pair of programs satisfy TI-GNI iff $\mathcal{C}_{\text{TIGNI}}$ is satisfiable.*

Theorem 4 (Soundess and Completeness of TS-GNI). *The given pair of programs satisfy TS-GNI iff $\mathcal{C}_{\text{TSGNI}}$ is satisfiable.*

The soundness directions are proven by "determinizing" the angelic choices by solutions to the functional predicate variables and reducing the argument to those of k-safety and co-termination. The completeness directions are proven by "synthesizing" sufficient angelic choice functions from program executions.

Example 3. Via the encoding, our PCSat tool is able to verify the TS-GNI example from Sec. 2.1 by automatically inferring not only the functional predicate described there but also relational invariants and well-founded relations given in the appendix. For space, the concrete constraint set is also given in the appendix.

Remark 3. The angelic non-determinism encoding can be optimized by using head disjunctions when the non-determinism is finitary (*i.e.*, $\max_{\widetilde{v}} |\{\widetilde{v}' \mid T_2(\widetilde{v}, \widetilde{v}')\}|$ is finite) instead of using functional predicate variables. For this, we modify clauses (3) and (3x)'s of Def. 7 and 8 to contain multiple positive occurrences of inv where each occurrence represents one of the finitely many possible choices.

Remark 4. Recall that we allow any program to be non-deterministic. The k-safety and co-termination encodings treat non-determinism in all programs as demonic, whereas the GNI encodings treat those in one program (*i.e.*, P_1) as demonic and those in the other program (*i.e.*, P_2) as angelic. In general, an arbitrary program can be made angelic by applying the transformation done in the angelic side of GNI encodings (to factor out non-determinism).

5 Constraint Solving Method for pfwCSP

We describe a CEGIS-based method for finding a (syntactic) solution of the given pfwCSP $(\mathcal{C}, \mathcal{K})$. Our method iterates the following phases until convergence. The iteration maintains and builds a sequence σ of *candidate solutions* and a sequence $\mathcal{E}$ of *example instances* where $\mathcal{E}^{(i)}$ are ground clauses obtained from $\mathcal{C}$ by instantiating the term variables and serve as a counterexample to the candidate solution $\sigma^{(i-1)}$, for each i-th iteration. The iteration starts from $\mathcal{E}^{(1)} = \emptyset$.

Synthesis Phase: We check if $(\mathcal{E}^{(i)}, \mathcal{K})$ is unsatisfiable. If so, we stop by returning $\mathcal{E}^{(i)}$ as a genuine counterexample to the input problem $(\mathcal{C}, \mathcal{K})$. Otherwise, we use the synthesizer $\mathcal{S}_{TB}$ (cf. Sec. 5.1) to find a solution $\sigma^{(i)}$ of $(\mathcal{E}^{(i)}, \mathcal{K})$, which will be used as the next candidate solution.

Validation Phase: We check if $\sigma^{(i)}$ is a genuine solution to $(\mathcal{C}, \mathcal{K})$ by using an SMT solver. If so, we stop by returning $\sigma^{(i)}$ as a solution. Otherwise, for each clause $c \in \mathcal{C}$ not satisfied by $\sigma^{(i)}$, we obtain a term substitution θ_c such that $\mathrm{dom}(\theta_c) = \mathit{ftv}(c)$ and $\not\models \theta_c(\sigma^{(i)}(c))$. We then update the example set by adding a new example instance for each unsatisfied clause (i.e., $\mathcal{E}^{(i+1)} = \mathcal{E}^{(i)} \cup \{\, \theta_c(c) \mid c \in \mathcal{C} \wedge \not\models \sigma^{(i)}(c) \,\}$), and proceed to the next iteration.

The above procedure satisfies the usual *progress property* of CEGIS: discovered counterexamples and candidate solutions are not discovered again in succeeding iterations. Furthermore, as discussed in Sec. 5.1, by carefully designing the synthesizer $\mathcal{S}_{TB}$ by incorporating *stratified* CEGIS, we achieve *completeness* in the sense of [34, 56]: if the given pfwCSP $(\mathcal{C}, \mathcal{K})$ has a syntactic solution expressible in the stratified families of templates, a solution of the pfwCSP is eventually found by the procedure. In Sec. 5.1, we discuss the details of the synthesis phase. There, for space, we focus on the theory of quantifier-free linear integer arithmetic (QFLIA). For space, we defer the details of the unsatisfiability checking process to Appendix K.

Remark 5. The implementation described in Sec. 6 contains an additional phase called *resolution phase* for accelerating the convergence. There, we first apply unit propagation repeatedly to the given $\mathcal{E}^{(i)}$ to obtain positive examples $\mathcal{E}^{(i)+}$ of the form $X(\widetilde{v})$ and negative examples $\mathcal{E}^{(i)-}$ of the form $\neg X(\widetilde{v})$. We then repeatedly apply resolution principle to the clauses in the input clauses $\mathcal{C}$ and the clauses $\mathcal{E}^{(i)+} \cup \mathcal{E}^{(i)-}$ to obtain additional positive and negative examples.

5.1 Predicate Synthesis with Stratified Families of Templates

We describe our candidate solution synthesizer $\mathcal{S}_{TB}$. $\mathcal{S}_{TB}$ performs a template-based search for a solution to the given example instances. As we shall show, our approach allows searching for assignments to all predicate variables (of all three kinds) in the given instance which is important because satisfying assignments to different predicate variables often inter-dependent. There, however, is a trade-off between expressiveness and generalizability. With less expressive templates like intervals, we may miss actual solutions. But with very expressive templates like polyhedra, there could be many solutions, and a solution thus returned is liable

Stratified Template Family for Ordinary Predicate Variables:

$$T^{\bullet}_X(nd, nc, ac, ad) \triangleq \lambda(x_1, \ldots, x_{\mathrm{ar}(X)}).\bigvee_{i=1}^{nd}\bigwedge_{j=1}^{nc} c_{i,j,0} + \sum_{k=1}^{\mathrm{ar}(X)} c_{i,j,k}\cdot x_k \geq 0$$
$$\phi^{\bullet}_X(nd, nc, ac, ad) \triangleq \bigwedge_{i=1}^{nd}\bigwedge_{j=1}^{nc}(\sum_{k=1}^{\mathrm{ar}(X)}|c_{i,j,k}| \leq ac) \wedge |c_{i,j,0}| \leq ad$$

Stratified Template Family for Well-Founded Predicate Variables:

$$T^{\Downarrow}_X(np, nl, nc, rc, rd, dc, dd) \triangleq \lambda(\widetilde{x}, \widetilde{y}).\bigwedge_{i=1}^{np}\bigwedge_{k=1}^{nl} r_{i,k}(\widetilde{x}) \geq 0 \wedge (\bigvee_{i=1}^{np} D_i(\widetilde{x})) \wedge (\bigvee_{j=1}^{np} D_j(\widetilde{y})) \wedge (\bigvee_{i=1}^{np} D_i(\widetilde{x}) \wedge \bigwedge_{j=1}^{np}(D_j(\widetilde{y}) \Rightarrow DEC_{i,j}(\widetilde{x}, \widetilde{y})))$$
$$\phi^{\Downarrow}_X(np, nl, nc, rc, rd, dc, dd) \triangleq \bigwedge_{i=1}^{np}\bigwedge_{k=1}^{nl}(\sum_{\ell=1}^{\mathrm{ar}(X)/2}|c_{i,k,\ell}| \leq rc) \wedge |c_{i,k,0}| \leq rd \wedge \bigwedge_{i=1}^{np}\bigwedge_{k=1}^{nc}(\sum_{\ell=1}^{\mathrm{ar}(X)/2}|c'_{i,k,\ell}| \leq dc) \wedge |c'_{i,k,0}| \leq dd$$
$$DEC_{i,j}(\widetilde{x}, \widetilde{y}) \triangleq \bigvee_{k=1}^{nl}(r_{i,k}(\widetilde{x}) > r_{j,k}(\widetilde{y}) \wedge \bigwedge_{\ell=1}^{k-1} r_{i,\ell}(\widetilde{x}) \geq r_{j,\ell}(\widetilde{y}))$$
$$r_{i,k}(\widetilde{x}) \triangleq c_{i,k,0} + \sum_{\ell=1}^{\mathrm{ar}(X)/2} c_{i,k,\ell}\cdot x_\ell \qquad D_i(\widetilde{x}) \triangleq \bigwedge_{k=1}^{nc} c'_{i,k,0} + \sum_{\ell=1}^{\mathrm{ar}(X)/2} c'_{i,k,\ell}\cdot x_\ell \geq 0$$

Stratified Template Family for Functional Predicate Variables:

$$T^{\lambda}_X(nd, nc, dc, dd, ec, ed) \triangleq \lambda(\widetilde{x}, r).r = \mathsf{if}\ D_1(\widetilde{x})\ \mathsf{then}\ e_1(\widetilde{x})\ \mathsf{else\ if}\ D_2(\widetilde{x})\ \mathsf{then}\ e_2(\widetilde{x}) \cdots \mathsf{else\ if}\ D_{nd-1}(\widetilde{x})\ \mathsf{then}\ e_{nd-1}(\widetilde{x})\ \mathsf{else}\ e_{nd}(\widetilde{x})$$
$$\phi^{\lambda}_X(nd, nc, ec, ed, dc, dd) \triangleq \bigwedge_{i=1}^{nd}(\sum_{j=1}^{\mathrm{ar}(X)-1}|c_{i,j}| \leq ec) \wedge |c_{i,0}| \leq ed \wedge \bigwedge_{i=1}^{nd-1}\bigwedge_{j=1}^{nc}(\sum_{k=1}^{\mathrm{ar}(X)-1}|c'_{i,j,k}| \leq dc) \wedge |c'_{i,j,0}| \leq dd$$
$$e_i(\widetilde{x}) \triangleq c_{i,0} + \sum_{j=1}^{\mathrm{ar}(X)-1} c_{i,j}\cdot x_j \qquad D_i(\widetilde{x}) \triangleq \bigwedge_{j=1}^{nc} c'_{i,j,0} + \sum_{k=1}^{\mathrm{ar}(X)-1} c'_{i,j,k}\cdot x_k \geq 0$$

Fig. 2. Stratified Families of Templates

to overfitting, *i.e.*, the solution to the example instances becomes too specific to be an actual solution to the original input clauses. [45] discusses a similar overfitting issue in the context of grammar-based synthesis.

Our remedy to the issue is *stratified families of predicate templates*, inspired by a similar approach proposed in the context of predicate abstraction with CEGAR [34, 56]. Initially, we assign each predicate variable a less expressive template and gradually refine it in a counterexample-guided manner: if no solution exists in the current template, we generate and analyze an unsat core to identify the *parameters of the families of templates* that should be updated. The stratification of templates thus automatically pushes the template to an expressive one (e.g., polyhedra) when it needs to. Importantly, with our approach, expressive templates are not always used but only when they should be used.

Stratified Families of Templates We have designed three stratified families of templates shown in Fig. 2, respectively for ordinary ($\bullet$), well-founded ($\Downarrow$), and functional (λ) predicate variables. First, for each ordinary predicate variable X, we prepare the stratified family of templates $T^{\bullet}_X(nd, nc, ac, ad)$ with unknowns $c_{i,j,k}$'s to be inferred and its accompanying constraint $\phi^{\bullet}_X(nd, nc, ac, ad)$. The body of $T^{\bullet}_X$ is a DNF with affine inequalities as atoms. The parameter nd (resp. nc) is the number of disjuncts (resp. conjuncts). The parameter ac is the upper bound of the sum of the absolute values of coefficients $c_{i,j,k}$ ($k > 0$), and ad is the upper bound of the absolute value of $c_{i,j,0}$.

Secondly, for each functional predicate variable X, we prepare the stratified family of templates $T^{\Downarrow}_X(np, nl, nc, rc, rd, dc, dd)$ with unknowns $c_{i,j,k}$'s and

$c'_{i,j,k}$'s and its accompanying constraint $\phi^{\Downarrow}_X(np, nl, nc, rc, rd, dc, dd)$. $T^{\Downarrow}_X$ represents the well-founded relation induced by a *piecewise-defined lexicographic affine ranking function* [2, 40, 40, 60, 61] where $r_{i,j}$ is the affine ranking function template for the j-th lexicographic component of the i-th region specified by the discriminator D_i. The parameter np (resp. nl) is the number of regions (resp. lexicographic components). The parameters rc, rd, dc, dd are the upper bounds of (the sums of) the absolute values of unknowns, similar to ac and ad of $T^{\bullet}_X$. The first conjunct of $T^{\Downarrow}_X$ asserts that the return value of each ranking functions is non-negative. The second and the third conjuncts assert that the discriminators cover all relevant states. Note that discriminators may overlap, and for such overlapping regions, the maximum return value of the ranking functions is used. The fourth conjunct asserts that the return value of the piecewise-defined ranking function strictly decreases from $\widetilde{x}$ to $\widetilde{y}$. Here, $DEC_{i,j}(\widetilde{x}, \widetilde{y})$ asserts that the return value of the lexicographic ranking function for the i-th region at $\widetilde{x}$ is greater than that for the j-th region at $\widetilde{y}$. It follows that for any substitution θ for the unknowns in $T^{\Downarrow}_X$, $\theta(T^{\Downarrow}_X)$ represents a well-founded relation. Our implementation PCSAT uses a refined version of $T^{\Downarrow}_X$ shown in Appendix L.

Finally, for each functional predicate variable X, we prepare the stratified family of templates $T^{\lambda}_X(nd, nc, dc, dd, ec, ed)$ with unknowns $c_{i,j}$'s and $c'_{i,j,k}$'s and its accompanying constraint $\phi^{\lambda}_F(nd, nc, dc, dd, ec, ed)$. T^{λ}_X characterizes a piecewise-defined affine function with discriminators $D_1, \ldots, D_{nd-1}$ and branch expressions $e_1, \ldots, e_{nd}$. The parameter nc is the number of conjuncts in each discriminator. The parameters dc, dd, ec, ed are the upper bounds of (the sums of) the absolute values of unknown, similar to ac and ad of $T^{\bullet}_X$. Note that for any substitution θ for the unknowns in T^{λ}_X, $\theta(T^{\lambda}_X)(\widetilde{x}, r)$ expresses a total function that maps $\widetilde{x}$ to r.

Next, we give the details of the candidate solution synthesis process. Let $\widetilde{p} \in \mathbb{Z}^n$ where n is the number of parameters summed across all templates, and let $T^{\alpha}_X(\widetilde{p})$ and $\phi^{\alpha}_X(\widetilde{p})$ (for $\alpha \in \{\bullet, \Downarrow, \lambda\}$) project the corresponding parameters. Each $\widetilde{p} \in \mathbb{Z}^n$ induces a *solution space* $[\![\widetilde{p}]\!] \triangleq \{T(\widetilde{p})[\theta] \mid \theta \models Con(\widetilde{p})\}$ where $T(\widetilde{p})[\theta] \triangleq \{X \mapsto \theta(T^{\mathcal{K}(X)}_X(\widetilde{p})) \mid X \in fpv(\mathcal{C})\}$ and $Con(\widetilde{p}) \triangleq \bigwedge_{X \in fpv(\mathcal{C})} \phi^{\mathcal{K}(X)}_X(\widetilde{p})$.

Let $\widetilde{p_1} \leq \widetilde{p_2}$ be the point-wise ordering. Note that $[\![\widetilde{p}]\!]$ is a finite set for any $\widetilde{p} \in \mathbb{Z}^n$, and $\widetilde{p_1} \leq \widetilde{p_2}$ implies $[\![\widetilde{p_1}]\!] \subseteq [\![\widetilde{p_2}]\!]$. We start the CEGIS process with some small initial parameters $\widetilde{p}^{(0)}$ (the parameters will be maintained as a state of the CEGIS process). The synthesis phase of each iteration tries to find a solution $\theta \in [\![\widetilde{p}^{(i)}]\!]$ to the given example instances $(\mathcal{E}, \mathcal{K})$ where $\widetilde{p}^{(i)}$ are the current parameters. This is done by using an SMT solver for QFLIA to find θ satisfying $\bigwedge T(\widetilde{p}^{(i)})[\theta](\mathcal{E}) \wedge \theta(Con(\widetilde{p}^{(i)}))$. If such θ is found, we return $T(\widetilde{p}^{(i)})[\theta]$ as the candidate solution for the next validation phase of the CEGIS process. Note that, by construction of the templates, the solution is guaranteed to assign each well-founded (resp. functional) predicate variable a well-founded relation (resp. total function). Otherwise, no solutions to the given example instances $(\mathcal{E}, \mathcal{K})$ can be found in $[\![\widetilde{p}^{(i)}]\!]$, and we update the parameters to some $\widetilde{p}^{(i+1)} > \widetilde{p}^{(i)}$ such that $[\![\widetilde{p}^{(i+1)}]\!]$ contains a solution for $(\mathcal{E}, \mathcal{K})$. Here, it is important to do the update in a *fair manner* [34,56], that is, in any infinite series of updates

$\widetilde{p}^{(0)}, \widetilde{p}^{(1)}, \ldots$, every parameter is updated infinitely often (the details are deferred to below). By the progress property and the fact that every $[\![\widetilde{p}]\!]$ is finite, this ensures that every parameter is updated infinitely often in an infinite series of CEGIS iterations. We thus obtain the following property.

Theorem 5. *Our CEGIS-procedure based on stratified families of templates is complete in the sense of [34, 56]: if there is* $\widetilde{p}$ *and* $\sigma \in [\![\widetilde{p}]\!]$ *such that* σ *is a syntactic solution to the given* pfwCSP $(\mathcal{C}, \mathcal{K})$*, a syntactic solution to* $(\mathcal{C}, \mathcal{K})$ *is eventually found by the procedure.*

Note that, while the solution space of each stratum (*i.e.*, $[\![\widetilde{p}^{(i)}]\!]$) is finite, our procedure searches the infinite solution space obtained by taking the infinite union of the solution spaces of the template family strata (*i.e.*, $\bigcup_{i\in\omega} [\![\widetilde{p}^{(i)}]\!]$).

Remark 6. Our template-based synthesis simultaneously finds ordinary, well-founded, and functional predicates that are mutually dependent through the given $(\mathcal{E}, \mathcal{K})$. This means that templates for different kinds of predicate variables are updated in a synchronized and balanced manner, which benefits the synthesis of mutually dependent witnesses for a relational property (see Appendix B, D, and F for examples).

Updating Parameters of Template Families via Unsat Cores. We now describe the parameter update process. We first obtain the unsat core of the unsatisfiability of $\bigwedge T(\widetilde{p}^{(i)})[\theta](\mathcal{E}) \wedge \theta(\mathit{Con}(\widetilde{p}^{(i)}))$ from the SMT solver. We then analyze the core to obtain the parameters of template families, such as the number of conjuncts and disjuncts, that have caused the unsatisfiability. Here, there could be a dependency between predicate variables and in such a case our unsat core analysis enumerates all the involved predicate variables from which we obtain the parameters of template families to be updated. We then increment these parameters in some fair manner, by limiting the maximum differences between different parameters to some bounded threshold, and repeatedly solve the resulting constraint until a solution is found. Thus, the parameters of stratified families of templates are grown on-the-fly guided by the reasons for unsatisfiability. We found that a careful design of parameter update strategies important for scaling the stratified CEGIS to hard relational verification problems. The manual tuning, however, is tiresome and suboptimal. We leave as future work to investigate methods for automatic tuning of parameter update strategies.

6 Evaluation

To evaluate the presented verification framework, we have implemented PCSat, a satisfiability checking tool for pfwCSP based on stratified CEGIS. PCSat supports the theory of Booleans and the quantifier-free theory of linear inequalities over integers and rationals. The tool is implemented in OCaml, using Z3 [42] as the backend SMT solver. We ran the tool on a diverse collection of 20 relational verification problems, consisting of k-safety, co-termination, and GNI problems.

Though we have manually reduced them to pfwCSP using the presented method in Sec. 4, this process can be easily automated. The full benchmark set is provided in Appendix M. All experiments have been conducted on 3.1GHz Intel Xeon Platinum 8000 CPU and 32 GiB RAM with the time limit of 600 seconds.

The experimental results are summarized in Table 1. The columns "Time (s)" and "#Iters" respectively show elapsed wall clock time in seconds and numbers of CEGIS iterations. PCSat solved 15 verification problems fully automatically and 5 problems labeled with the symbol † and/or ‡ semi-automatically. For the 4 problems labeled with †, we manually provided small hints for invariant synthesis (interested readers are referred to Appendix M). The provided hints for all but `SquareSum` are non-relational invariants that can be inferred prior to relational verification by using a CHCs solver or an invariant synthesizer. For the 2 problems labeled with ‡, we manually chose the initial value for the parameters of the template family for ordinary predicate variables to reduce the elapsed time. This can be automated by running PCSat with different initial values in parallel.

The problems `DoubleSquareNI_h**`, `HalfSquareNI`, `ArrayInsert`, and `SquareSum` are k-safety verification problems obtained from [51] that require non-lock-step scheduling.[7] The problems `DoubleSquareNI_h**` are generated from Example 1 by a case analysis of the valuation for the boolean variables h_1 and h_2. PCSat solved all the k-safety problems but `SquareSum` *fully automatically*. The tool Pdsc proposed in [51] can verify them but requires the user to provide the atomic predicates for expressing relational invariants and schedulers. The problems `CotermIntro1` and `CotermIntro2` are asymmetric co-termination problems obtained from the symmetric problem in Example 2 and are verified by PCSat fully automatically. The problems `TS_GNI_h**` are generated from Example 3 by a case analysis and are verified by PCSat with small non-relational hints. We have also tested PCSat on various TS-GNI (`SimpleTS_GNI1`, `SimpleTS_GNI2`, `InfBranchTS_GNI`) and TI-GNI problems (`TI_GNI_h**`) and obtained promising results. As far as we know, no existing tools can verify these non-k-safety relational problems.

Furthermore, manual inspection of the PCSat's output logs for the GNI problems that required hints revealed that the functional predicate synthesis appears to be the main bottleneck of the current version. In fact, we confirmed that the problems can be solved in less than 10 seconds if appropriate functional predicates for angelic non-determinism are manually provided. As future work, we plan to investigate methods for improved functional predicate synthesis.

7 Related Work

7.1 Relational Verification

There has been substantial work on verifying relational properties. They include program logics, type systems, or program analysis frameworks such as abstract

[7] We omitted `ArrayIntMod` from [51] because its verification requires the theory of arrays which the current version of PCSat does not fully support.

Table 1. Experimental results of PCSat on the relational verification benchmarks

Program	Time (s)	#Iters	Program	Time (s)	#Iters
DoubleSquareNI_hFT	17.762	42	HalfSquareNI	11.853	35
DoubleSquareNI_hTF	26.495	55	ArrayInsert‡	118.671	73
DoubleSquareNI_hFF	2.944	9	SquareSum†‡	337.596	117
DoubleSquareNI_hTT	4.055	11	SimpleTS_GNI1	5.397	14
CotermIntro1	19.322	80	SimpleTS_GNI2	8.919	26
CotermIntro2	15.871	73	InfBranchTS_GNI	2.607	4
TS_GNI_hFT†	47.083	78	TI_GNI_hFT†	4.389	16
TS_GNI_hTF	5.076	17	TI_GNI_hTF	2.277	6
TS_GNI_hFF	7.174	24	TI_GNI_hFF	2.968	6
TS_GNI_hTT†	23.495	53	TI_GNI_hTT	4.148	22

interpretation and model checking [1,5,9,19,25,53,62], program transformation approaches such as self-composition or product programs [4,7,15,20,21,43,48,55, 58,64], and various other approaches [3,18,23,47,59]. We refer to [44] for an excellent survey. However, most prior automatic approaches address only the k-safety fragment [17,55] and cannot verify non-k-safety (actually, not even hypersafety) properties such as co-termination, TS-NI, TI-GNI, and TS-GNI [6, 11, 41]. The only exception that we are aware is the recent work by Coenen et al. [19] that proposes a sound method for automatically verifying $\forall\exists$ hyperproperties such as GNI for finite state systems. To our knowledge, we are the *first* to propose a sound-and-complete approach to automatically verifying these non-hypersafety properties for infinite state programs.[8]

A key task in many relational verification methods, including ours, is the discovery of *relational invariants* which relate the states of multiple program executions. While most prior approaches are limited to fixed execution schedule (or *alignment*) such as lock-step and sequential [7, 8, 20, 21, 43, 55, 58], a recent work by Shemer et al. [51] has proposed a k-safety property verification method that automatically infers fair schedulers sufficient to prove the goal property. Importantly, the schedulers in their approach can be *semantic* in which the choice of which program to execute can depend on the *states* of the programs as opposed to the classic *syntactic* schedulers such as lock-step and sequential that can only depend on the control locations. Our approach also infers such fair semantic schedulers, and as remarked before, they enable solving instances like doubleSquare that are difficult for previous approaches. However, [51] requires the user to provide appropriate atomic predicates and is not fully automatic. By contrast, our approach soundly and completely encodes the problem as a

[8] However, [19] can verify (relational) temporal properties, whereas we only support functional properties that are given by pre- and post-conditions of whole program runs. We leave as future work to investigate methods for verifying relational temporal properties of infinite state programs.

constraint satisfaction problem and fully automatically verifies hard instances like doubleSquare by constraint solving.

Furthermore, our work extends the fair semantic scheduler synthesis to beyond k-safety problems like co-termination, TI-GNI and TS-GNI, in a sound and complete manner. We note that the extensions are non-trivial and involves delicate uses of functional predicate variables and well-founded predicate variables to ensure scheduler fairness in the presence of non-termination as well as uses of prophecy variables and functional predicate variables to handle angelic non-determinism. The higher-degree of automation and the extension to non-k-safety properties are thanks to the expressive power of our novel constraint framework pfwCSP.

7.2 Predicate Constraint Solving

Our pfwCSP solving technique builds on and generalizes a number of techniques developed for CHCs solving as well as invariant and ranking function discovery. Most closely related to our constraint solving method are CEGIS-based [52] and data-driven approaches to solving CHCs [14,22,24,26,27,39,45,46,49,50,65]. As remarked before, the new pfwCSP framework is strictly more expressive than CHCs and extending the prior techniques to the new framework is non-trivial.

Our stratified CEGIS is inspired by the idea of stratified languages of predicates proposed in the context of predicate abstraction with CEGAR [34,56]. It is also similar in spirit to the work by Padhi et al. [45], but they use a stratified family of grammars. Also none of these prior works use unsat cores for updating the language/grammar stratum, synthesize well-founded relations and functional predicates, or support non-Horn clauses.

Our class of pfwCSP constraints is related to *existentially-quantified Horn clauses* (E-CHCs) introduced by Beyene et al. [12]. E-CHCs does not have non-Horn clauses or functional predicate variables. However, it has disjunctive well-foundedness constraints which are similar to our well-founded predicate variables. Also, existential quantifiers can be used to encode head disjunctions and functional predicates. We conjecture that pfwCSP and E-CHCs are inter-reducible, but it is not trivial to fill the gap. Also, inter-reducibility is a desirable feature: different formats may have different benefits. For relational verification, as we have shown, pfwCSP enables direct sound-and-complete encodings of the problems. For instance, head disjunctions allow direct encoding of scheduler fairness and finitary angelic non-determinism (cf. Remark 3). And, functional predicate variables can be explicitly given necessary-and-sufficient parameters to encode angelic non-determinism and difference bounds for ensuring scheduler fairness in the presence of non-termination. The tight encodings also lead to reduction in search space and benefited the constraint solving.

8 Conclusion

We have introduced the class pfwCSP of predicate constraint satisfaction problems that generalizes CHCs with arbitrary clauses, well-foundedness constraints,

and functionality constraints. We have then established a program verification framework based on pfwCSP by showing that (1) pfwCSP can soundly-and-completely encode various classes of relational problems of infinite-state non-deterministic programs, including hard instances of k-safety, co-termination, and termination-sensitive generalized non-interference that benefit from state-dependent scheduling/alignment (Theorems 1–4), and (2) existing CHCs solving and invariants/ranking function synthesis techniques can be adopted to pfwCSP solving and further improved with the idea of stratified CEGIS for simultaneously achieving completeness (Theorem 5) and efficiency.

In future work we plan to investigate ways to improve functional predicate synthesis, automatic tuning of parameter update strategies for constraint solving, and whether a constraint-based approach (and the techniques presented in the present paper) can be extended to reason about relational temporal properties such as the ones expressed in hyper temporal logics [16, 25].

Acknowledgments. We thank the anonymous reviewers for their suggestions. This work was supported by ONR grant # N00014-17-1-2787, JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), and JSPS KAKENHI Grant Numbers 17H01720, 18K19787, 19H04084, 20H04162, 20H05703, and 20K20625.

References

1. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. J. Funct. Program. **29** (2019)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: SAS '10. pp. 117–133. Springer (2010)
3. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: PLDI (2017)
4. Asada, K., Sato, R., Kobayashi, N.: Verifying relational properties of functional programs by first-order refinement. In: PEPM (2015)
5. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: POPL (2017)
6. Barthe, G.: An introduction to relational program verification (2020)
7. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM (2011)
8. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW (2004)
9. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
10. Beringer, L.: Relational bytecode correlations. J. Log. Alg. Meth. Pro. **79**(7) (2010)
11. Beringer, L., Hofmann, M.: Secure information flow and program logics. Arch. Formal Proofs (2008)
12. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: CAV (2013)

13. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (2015)
14. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: TACAS (2018)
15. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: PLDI (2019)
16. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: POST (2014)
17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: CSF (2008)
18. Clochard, M., Marché, C., Paskevich, A.: Deductive verification with ghost monitors. PACMPL **4**(POPL) (2020)
19. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: CAV (2019)
20. Ádám Darvas, Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: SPC (2005)
21. Eilers, M., Müller, P., Hitz, S.: Modular product programs. TOPLAS **42**(1) (2020)
22. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. PACMPL **2**(OOPSLA) (2018)
23. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: CAV (2019)
24. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-guided termination analysis. In: CAV (2018)
25. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: CAV (2015)
26. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV (2014)
27. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL (2016)
28. Gonnord, L., Monniaux, D., Radanne, G.: Synthesis of ranking functions using extremal counterexamples. In: PLDI (2015)
29. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
30. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV (2015)
31. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: CADE (2013)
32. Hojjat, H., Rümmer, P.: The Eldarica horn solver. In: FMCAD (2018)
33. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: verifying functional programs using abstract interpreters. In: CAV (2011)
34. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS (2006)
35. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. **4** (1975)
36. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: CAV (2016)
37. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI (2011)
38. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV (2014)
39. Krishna, S., Puhrsch, C., Wies, T.: Learning invariants using decision trees. CoRR **abs/1501.04725** (2015)

40. Leike, J., Heizmann, M.: Ranking templates for linear loops. LMCS **11**(1) (2015)
41. McCullough, D.: Noninterference and the composability of security properties. In: SP (1988)
42. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
43. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: ESORICS (2006)
44. Naumann, D.A.: Thirty-seven years of relational hoare logic: remarks on its principles and history. CoRR **abs/2007.06421** (2020)
45. Padhi, S., Millstein, T.D., Nori, A.V., Sharma, R.: Overfitting in synthesis: Theory and practice. In: CAV (2019)
46. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI (2016)
47. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: CAV (2018)
48. Reynolds, J.C.: The craft of programming. Prentice Hall (1981)
49. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP (2013)
50. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: SAS (2013)
51. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: CAV (2019)
52. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
53. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: PLDI (2016)
54. Terauchi, T.: Dependent types from counterexamples. In: POPL (2010)
55. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: SAS (2005)
56. Terauchi, T., Unno, H.: Relaxed stratification: A new approach to practical complete predicate refinement. In: ESOP (2015)
57. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP (2009)
58. Unno, H., Kobayashi, N., Yonezawa, A.: Combining type-based analysis and model checking for finding counterexamples against non-interference. In: PLAS (2006)
59. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: CAV (2017)
60. Urban, C.: The abstract domain of segmented ranking functions. In: SAS (2013)
61. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: ESOP (2014)
62. Volpano, D.M., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comp. Sec. **4**(2-3) (1996)
63. Volpano, D.M., Smith, G.: Eliminating covert flows with minimum typings. In: CSFW (1997)
64. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: FM (2008)
65. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI (2018)
66. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: ICFP (2015)

A The encoding of TI-NI verification from Example 1

$$
\begin{aligned}
\mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) &\Leftarrow x_1 = x_2 \wedge \\
&\quad\ y_1 = 0 \wedge (h_1 \wedge z_1 = 2 \times x_1 \vee \neg h_1 \wedge z_1 = x_1) \wedge \\
&\quad\ y_2 = 0 \wedge (h_2 \wedge z_2 = 2 \times x_2 \vee \neg h_2 \wedge z_2 = x_2) \\
\mathsf{inv}(\widetilde{V}_1', \widetilde{V}_2) &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TF}}(\widetilde{V}_1, \widetilde{V}_2) \wedge \\
&\quad\ (z_1 > 0 \wedge z_1' = z_1 - 1 \wedge y_1' = y_1 + x_1 \vee z_1 \leq 0 \wedge z_1' = z_1 \wedge y_1' = y_1) \\
\mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2') &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{FT}}(\widetilde{V}_1, \widetilde{V}_2) \wedge \\
&\quad\ (z_2 > 0 \wedge z_2' = z_2 - 1 \wedge y_2' = y_2 + x_2 \vee z_2 \leq 0 \wedge z_2' = z_2 \wedge y_2' = y_2) \\
\mathsf{inv}(\widetilde{V}_1', \widetilde{V}_2') &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TT}}(\widetilde{V}_1, \widetilde{V}_2) \wedge \\
&\quad\ (z_1 > 0 \wedge z_1' = z_1 - 1 \wedge y_1' = y_1 + x_1 \vee z_1 \leq 0 \wedge z_1' = z_1 \wedge y_1' = y_1) \wedge \\
&\quad\ (z_2 > 0 \wedge z_2' = z_2 - 1 \wedge y_2' = y_2 + x_2 \vee z_2 \leq 0 \wedge z_2' = z_2 \wedge y_2' = y_2) \\
z_1 > 0 &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TF}}(\widetilde{V}_1, \widetilde{V}_2) \wedge z_2 > 0 \\
z_2 > 0 &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{FT}}(\widetilde{V}_1, \widetilde{V}_2) \wedge z_1 > 0 \\
\mathsf{sch}_{\mathsf{TF}}(\widetilde{V}_1, \widetilde{V}_2) \vee \mathsf{sch}_{\mathsf{FT}}(\widetilde{V}_1, \widetilde{V}_2) \vee \mathsf{sch}_{\mathsf{TT}}(\widetilde{V}_1, \widetilde{V}_2) &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge (z_1 > 0 \vee z_2 > 0) \\
y_1' = y_2' &\Leftarrow \mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \wedge z_1 \leq 0 \wedge z_2 \leq 0 \wedge \\
&\quad\ (h_1 \wedge y_1' = y_1 \vee \neg h_1 \wedge y_1' = 2 \times y_1) \wedge \\
&\quad\ (h_2 \wedge y_2' = y_2 \vee \neg h_2 \wedge y_2' = 2 \times y_2)
\end{aligned}
$$

where $\widetilde{V}_1 = x_1, y_1, z_1, h_1$, $\widetilde{V}_1' = x_1, y_1', z_1', h_1'$, $\widetilde{V}_2 = x_2, y_2, z_2, h_2$, $\widetilde{V}_2' = x_2, y_2', z_2', h_2'$, $\mathsf{TF} = \{1\}$, $\mathsf{FT} = \{2\}$, and $\mathsf{TT} = \{1, 2\}$.

B The PCSat generated solution of Example 1

$$
\mathsf{inv}(\widetilde{V}_1, \widetilde{V}_2) \equiv \left(\begin{array}{l}
\neg h_1 \wedge \neg h_2 \wedge \left(\begin{array}{l} x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2 \vee \\ x_2 + y_2 = 0 \wedge z_1 < 0 \wedge \\ 1 + 2 \cdot x_2 = 2 \cdot z_2 \wedge 2 + y_1 = y_2 \end{array}\right) \vee \\
\neg h_1 \wedge h_2 \wedge \left(\begin{array}{l} x_1 = x_2 \wedge 1 + 2 \cdot x_1 \neq z_2 \wedge 2 \cdot y_1 = y_2 \wedge 2 \cdot z_1 = z_2 \vee \\ x_1 = x_2 \wedge 2 \cdot x_1 \neq z_2 \wedge z_1 \geq 1 \wedge z_2 \geq 1 \wedge \\ x_1 + 2 \cdot y_1 = y_2 \wedge 2 \cdot z_1 = 1 + z_2 \end{array}\right) \vee \\
h_1 \wedge \neg h_2 \wedge \left(\begin{array}{l} x_1 = x_2 \wedge y_1 = 2 \cdot y_2 \wedge z_1 = 2 \cdot z_2 \vee \\ x_1 = x_2 \wedge 2 \cdot x_1 \neq z_1 \wedge z_1 \geq 1 \wedge z_2 \geq 1 \wedge \\ y_1 = x_2 + 2 \cdot y_2 \wedge 1 + z_1 = 2 \cdot z_2 \end{array}\right) \vee \\
h_1 \wedge h_2 \wedge x_1 = x_2 \wedge y_1 = y_2 \wedge z_2 = z_2
\end{array}\right)
$$

C The encoding of the co-termination verification from Example 2

$$
\begin{aligned}
\mathsf{inv}(0,b,\widetilde{V}_1,\widetilde{V}_2) &\Leftarrow \mathsf{fnb}(\widetilde{V}_1,\widetilde{V}_2,b)\wedge x_1=x_2\wedge y_1=y_2\\
\mathsf{inv}(d',b,\widetilde{V}_1',\widetilde{V}_2) &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge \mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge\\
&\qquad (x_1>0\wedge x_1'=x_1-y_1\vee x_1\le 0\wedge x_1'=x_1)\wedge\\
&\qquad (x_1\le 0\vee x_2\le 0\vee d'=d+1)\\
\mathsf{inv}(d',b,\widetilde{V}_1,\widetilde{V}_2') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge\\
&\qquad (x_2>0\wedge x_2'=x_2-2\cdot y_2\vee x_2\le 0\wedge x_2'=x_2)\wedge\\
&\qquad (x_1\le 0\vee x_2\le 0\vee d'=d-1)\\
\mathsf{inv}(d,b,\widetilde{V}_1',\widetilde{V}_2') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge \mathsf{sch}_{\mathsf{TT}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge\\
&\qquad (x_1>0\wedge x_1'=x_1-y_1\vee x_1\le 0\wedge x_1'=x_1)\wedge\\
&\qquad (x_2>0\wedge x_2'=x_2-2\cdot y_2\vee x_2\le 0\wedge x_2'=x_2)\\
x_1>0 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge \mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge x_2>0\\
x_2>0 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge x_1>0\\
\mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\vee \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2)\vee \mathsf{sch}_{\mathsf{TT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge(x_1>0\vee x_2>0)\\
-b\le d\wedge d\le b\wedge b\ge 0 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge x_1>0\wedge x_2>0\\
\mathsf{wfr}_1(\widetilde{V}_1,\widetilde{V}_1') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge x_2\le 0\wedge x_1>0\wedge x_1'=x_1-y_1\\
\mathsf{wfr}_2(\widetilde{V}_2,\widetilde{V}_2') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2)\wedge x_1\le 0\wedge x_2>0\wedge x_2'=x_2-2\cdot y_2
\end{aligned}
$$

where $\widetilde{V}_1 = x_1, y_1$, $\widetilde{V}_1' = x_1', y_1$, $\widetilde{V}_2 = x_2, y_2$, $\widetilde{V}_2' = x_2', y_2$, $\mathsf{TF} = \{1\}$, $\mathsf{FT} = \{2\}$, and $\mathsf{TT} = \{1,2\}$.

D The PCSat generated solution of Example 2

$$
\begin{aligned}
\mathsf{fnb}(x_1,y_1,x_2,u_2,b) &\equiv b=1\\
\mathsf{inv}(d,b,x_1,y_1,x_2,y_2) &\equiv d=0\wedge b\ge 0\wedge b\le 1\wedge \begin{pmatrix} x_1=x_2\wedge y_1=y_2\vee\\ y_1=y_2\wedge x_1+y_1\ge 1\wedge x_2+2\cdot y_2\ge 1\end{pmatrix}\\
\mathsf{wfr}_1(x,y,x',y') &\equiv \begin{pmatrix} x-1\ge 0\wedge x-1>x'-1\wedge\\ ((x'>0\wedge y'\le 0)\Rightarrow x-1\ge 0\wedge x-1>1-y')\vee\\ (x>0\wedge y\le 0)\wedge 1-y\ge 0\wedge 1-y>x'-1\wedge\\ ((x'>0\wedge y'\le 0)\Rightarrow 1-y\ge 0\wedge 1-y>1-y')\end{pmatrix}\\
\mathsf{wfr}_2(x,y,x',y') &\equiv \begin{pmatrix} (y'\ge 0\vee x'>0\wedge y'\ge 0)\wedge\\ (y\ge 0\wedge (y'\ge 0\Rightarrow -y\ge 0\wedge -y>-y')\wedge\\ \begin{pmatrix}(x'>0\wedge y'\ge 0)\Rightarrow -y\ge 0\wedge -y>x')\vee\\ (x>0\wedge y\ge 0)\wedge (y'\ge 0\Rightarrow x\ge 0\wedge x>-y')\wedge\\ ((x'>0\wedge y'\ge 0)\Rightarrow x\ge 0\wedge x>x')\end{pmatrix}\end{pmatrix}
\end{aligned}
$$

Here, wfr_1 is induced by the ranking function $\max(x-1, \texttt{if } x>0\wedge y\le 0 \texttt{ then } 1-y \texttt{ else } -\infty)$ and wfr_2 is induced by the ranking function $\max(\texttt{if } y\ge 0 \texttt{ then } -y \texttt{ else } -\infty, \texttt{if } x>0\wedge y\ge 0 \texttt{ then } x \texttt{ else } -\infty)$.

E The encoding of the TS-GNI verification from Example 3

$$
\begin{aligned}
\mathsf{inv}(0,b,\widetilde{V}_1,\widetilde{V}_2) &\Leftarrow \mathsf{fnb}(x_1,h_1,l_1,x_2,h_2,l_2,b) \wedge b_1 \wedge b_2 \wedge l_1 = l_2 \wedge \\
&\quad (h_1 \wedge x_1 = n_1 \vee \neg h_1 \wedge x_1 = l_1) \wedge \\
&\quad (h_2 \wedge \mathsf{fnr}(p,h_2,l_2,x_2) \vee \neg h_2 \wedge x_2 = l_2) \\
\mathsf{inv}(d',b,\widetilde{V}_1',\widetilde{V}_2) &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \\
&\quad \left(\begin{array}{l} b_1 \wedge h_1 \wedge (x_1 \geq l_1 \wedge \neg b_1' \wedge x_1' = x_1 \vee x_1 < l_1 \wedge b_1' \wedge x_1' = x_1) \vee \\ b_1 \wedge \neg h_1 \wedge (b_1' \wedge x_1' = x_1 + 1 \vee \neg b_1' \wedge x_1' = x_1) \vee \\ \neg b_1 \wedge \neg b_1' \wedge x_1' = x_1 \end{array}\right) \wedge \\
&\quad (\neg b_1 \vee \neg b_2 \vee d' = d+1) \\
\mathsf{inv}(d',b,\widetilde{V}_1,\widetilde{V}_2') \vee \mathsf{inv}(d',b,\widetilde{V}_1,\widetilde{V}_2'') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \\
&\quad \left(\begin{array}{l} b_2 \wedge h_2 \wedge \left(\begin{array}{l} x_2 \geq l_2 \wedge \neg b_2' \wedge x_2' = x_2 \wedge \neg b_2'' \wedge x_2'' = x_2 \vee \\ x_2 < l_2 \wedge b_2' \wedge x_2' = x_2 \wedge b_2'' \wedge x_2' = x_2 \end{array}\right) \vee \\ b_2 \wedge \neg h_2 \wedge (b_2' \wedge x_2' = x_2 + 1 \wedge \neg b_2'' \wedge x_2'' = x_2) \vee \\ \neg b_2 \wedge \neg b_2' \wedge x_2' = x_2 \wedge \neg b_2'' \wedge x_2'' = x_2 \end{array}\right) \wedge \\
&\quad (\neg b_1 \vee \neg b_2 \vee d' = d-1) \\
\mathsf{inv}(d,b,\widetilde{V}_1',\widetilde{V}_2') \vee \mathsf{inv}(d,b,\widetilde{V}_1',\widetilde{V}_2'') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \\
&\quad \left(\begin{array}{l} b_1 \wedge h_1 \wedge (x_1 \geq l_1 \wedge \neg b_1' \wedge x_1' = x_1 \vee x_1 < l_1 \wedge b_1' \wedge x_1' = x_1) \vee \\ b_1 \wedge \neg h_1 \wedge (b_1' \wedge x_1' = x_1 + 1 \vee \neg b_1' \wedge x_1' = x_1) \vee \\ \neg b_1 \wedge \neg b_1' \wedge x_1' = x_1 \end{array}\right) \wedge \\
&\quad \left(\begin{array}{l} b_2 \wedge h_2 \wedge \left(\begin{array}{l} x_2 \geq l_2 \wedge \neg b_2' \wedge x_2' = x_2 \wedge \neg b_2'' \wedge x_2'' = x_2 \vee \\ x_2 < l_2 \wedge b_2' \wedge x_2' = x_2 \wedge b_2'' \wedge x_2' = x_2 \end{array}\right) \vee \\ b_2 \wedge \neg h_2 \wedge (b_2' \wedge x_2' = x_2 + 1 \wedge \neg b_2'' \wedge x_2'' = x_2) \vee \\ \neg b_2 \wedge \neg b_2' \wedge x_2' = x_2 \wedge \neg b_2'' \wedge x_2'' = x_2 \end{array}\right) \\
b_1 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge b_2 \\
b_2 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge b_1 \\
\mathsf{sch}_{\mathsf{TF}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \vee \mathsf{sch}_{\mathsf{FT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) \vee \mathsf{sch}_{\mathsf{TT}}(d,b,\widetilde{V}_1,\widetilde{V}_2) &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge (b_1 \vee b_2) \\
-b \leq d \wedge d \leq b \wedge b \geq 0 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge b_1 \wedge b_2 \\
x_1 = x_2 &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \neg b_1 \wedge \neg b_2 \wedge p = x_1 \\
\mathsf{wfr}(\widetilde{V}_2,\widetilde{V}_2') &\Leftarrow \mathsf{inv}(d,b,\widetilde{V}_1,\widetilde{V}_2) \wedge \neg b_1 \wedge p = x_1 \wedge b_2 \wedge h_2 \wedge x_2 < l_2 \wedge x_2' = x_2
\end{aligned}
$$

where $\widetilde{V}_1 = p, b_1, x_1, h_1, l_1$, $\widetilde{V}_1' = p, b_1', x_1', h_1, l_1$, $\widetilde{V}_2 = b_2, x_2, h_2, l_2$, $\widetilde{V}_2' = b_2', x_2', h_2, l_2$, $\widetilde{V}_2'' = b_2'', x_2'', h_2, l_2$, $\mathsf{TF} = \{1\}$, $\mathsf{FT} = \{2\}$, and $\mathsf{TT} = \{1,2\}$.

F The PCSAT generated solution of Example 3

$$
\begin{aligned}
\mathsf{fnb}(x_1, y_1, x_2, u_2, b) &\equiv b = 0 \\
\mathsf{fnr}(p, h_2, l_2, x_2) &\equiv x_2 = p \\
\mathsf{inv}(d, b, \widetilde{V}_1, \widetilde{V}_2) &\equiv \left(\begin{array}{l}
\neg h_1 \wedge \neg h_2 \wedge d = 0 \wedge b = 0 \wedge b_2 \wedge x_2 \geq l_2 \wedge l_1 = l_2 \vee \\
\neg h_1 \wedge h_2 \wedge d = 0 \wedge b \geq 0 \wedge x_1 \geq l_1 \wedge p = x_2 \wedge l_1 = l_2 \vee \\
h_1 \wedge \neg h_2 \wedge \left(\begin{array}{l} d = 0 \wedge b = 0 \wedge b_2 \wedge l_1 = l_2 \vee \\ l_1 = x_2 \wedge p = x_1 \wedge x_1 \geq l_1 \wedge d \geq 1 + b \wedge \\ b = l_2 \wedge 1 + 2 \cdot x_1 + 2 \cdot x_2 = 0 \end{array}\right) \vee \\
h_1 \wedge h_2 \wedge \left(\begin{array}{l} d = 0 \wedge b = 0 \wedge b_1 \wedge l_1 = l_2 \wedge x_2 = p \vee \\ \neg b_1 \wedge x_1 \geq l_1 \wedge p = x_2 \wedge l_1 = l_2 \end{array}\right)
\end{array}\right) \\
\mathsf{wfr}(x, h, l, x', h', l') &\equiv \neg h \wedge l - x \geq 0 \wedge l - x > l' - x' \vee h \wedge x \geq 0 \wedge x > x'
\end{aligned}
$$

where $\widetilde{V}_1 = p, b_1, x_1, h_1, l_1$ and $\widetilde{V}_2 = b_2, x_2, h_2, l_2$.

G Proof of Theorem 1

Proof:

(only-if)

The only-if direction holds from the completeness of the standard lock-step product program construction that executes each programs synchronously in parallel. That is, such a product program is realized by the the scheduler defined by $\mathsf{sch}_{[k]} = \mathtt{true}$ and $\mathsf{sch}_A = \mathtt{false}$ for all $A \neq [k]$. Note that clauses (4) and (5) are trivially satisfied by such a scheduler. The corresponding inv is the set of tuples of states reachable by the lock-step evaluation from the tuples of states satisfying *Pre* (or any invariant used to verify the input instance under the lock-step product program construction). It is easy to see that such inv satisfies the rest of the clauses with the scheduler. (A similar argument can also be made with the standard sequential product program construction.)

(if)

We prove the if direction by proving the contrapositive. So, suppose that the tuple of programs violates the k-safety property. Then, there must be sequences $\pi_1, \ldots, \pi_k$ such that (a) $\mathit{Pre}(\pi_1[1], \ldots, \pi_k[1])$ is true, (b) for each π_i, $F_i(\pi_i[|\pi_i|])$ is true, (c) $\neg \mathit{Post}(\pi_1[|\pi_1|], \ldots, \pi_k[|\pi_k|])$ is true, and (d) for each π_i and $1 < j \leq |\pi_i|$, $T_i(\pi_i[j-1], \pi_i[j])$ is true. The following argument, roughly, says that under any scheduler satisfying $\mathcal{C}_{\mathrm{S}}$, we can "reach" $\pi_1[|\pi_1|], \ldots, \pi_k[|\pi_k|]$ from $\pi_1[1], \ldots, \pi_k[1]$. Let $a_i = 1$ for each $i \in [k]$. Let $\widetilde{v} = \pi_1[a_1], \ldots, \pi_k[a_k]$. By (a) and clause (1), it must be the case that $\mathsf{inv}(\widetilde{v})$ is true.

If all programs have terminated (i.e., $F_i(\pi_i[a_i])$ for each i), then by (b), (c) and the fact that programs self-loop after reaching final states, $\widetilde{v}$ invalidates clause (3) and therefore we have shown that $\mathcal{C}_{\mathrm{S}}$ is unsatisfiable. So, suppose that there are unfinished programs (i.e., $\bigvee_{i \in [k]} \neg F_i(\pi_i[a_i])$ is true). By clause

(5), there must be some $A \in \mathcal{P}^+[k]$ such that $\mathsf{sch}_A(\widetilde{v})$ is true. Then, by clause (4), there must be unfinished programs that A schedules to be evaluated next, that is, $B = \{i \in A \mid \neg F_i(\pi_i[a_i])\}$ is non-empty. Let $a'_i = a_i + 1$ if $i \in B$ and otherwise let $a'_i = a_i$. Let $\widetilde{v}' = \pi_1[a'_1], \ldots, \pi_k[a'_k]$. Then, $\mathsf{inv}(\widetilde{v}')$ must be true by (d) and clause (2).

Repeating the argument in the above paragraph by setting $\widetilde{v}$ to $\widetilde{v}'$ and a_i to a'_i for each $i \in [k]$, we will eventually reach the terminal tuple of states $\pi_1[|\pi_1|], \ldots, \pi_k[|\pi_k|]$ and show that $\mathsf{inv}(\pi_1[|\pi_1|], \ldots, \pi_k[|\pi_k|])$ must be true, which, by (c), invalidates clause (3) and therefore $\mathcal{C}_{\mathrm{S}}$ is unsatisfiable.

■

H Proof of Theorem 2

Proof:

(only-if)

Suppose that given pair of programs co-terminate. Let $\mathsf{sch}_{\mathsf{TT}} = \mathtt{true}$ and $\mathsf{sch}_{\mathsf{FT}} = \mathsf{sch}_{\mathsf{TF}} = \mathtt{false}$, *i.e.*, let the scheduler be lock-step. Let $\mathsf{fnb}(\widetilde{V}, b) = b = 0$ (any other predicate that sets b to be non-negative also works). Let inv be the set of tuples $(d, 0, \widetilde{v_1}, \widetilde{v_2})$ where $\widetilde{v_1}, \widetilde{v_2}$ are the tuples of states reachable from *Pre* by lock-step evaluation and $d = 0$ if $\neg F_1(\widetilde{v_1}) \wedge \neg F_2(\widetilde{v_2})$ (d is arbitrary if $F_1(\widetilde{v_1}) \vee F_2(\widetilde{v_2})$). Let R be the set of states of P_2 reachable from a pair of states satisfying *Pre* after the corresponding execution of P_1 has terminated by lock-step evaluation. That is, R is the set of states $\widetilde{v_2}$ satisfying the following: there exist a pair of states $(\widetilde{v_1}', \widetilde{v_2}')$ and a state $\widetilde{v_1}$ such that $\mathit{Pre}(\widetilde{v_1}', \widetilde{v_2}')$ is true, $(\widetilde{v_1}, \widetilde{v_2})$ is reachable from $(\widetilde{v_1}', \widetilde{v_2}')$ by lock-step evaluation, and $F_1(\widetilde{v_1})$ is true. Let $\mathsf{wfr} = (R \times R) \cap \{(\widetilde{v}, \widetilde{v}') \mid T_2(\widetilde{v}, \widetilde{v}')\}$ (or any other well-founded relation witnessing the termination of R). It is easy to see that these predicates satisfy $\mathcal{C}_{\mathrm{CoT}}$.

(if)

We prove the if direction by proving the contrapositive. So, suppose that the pair of programs violates co-termination. Then, there must be states $\widetilde{v_1}$,$\widetilde{v_2}$, and $\widetilde{v_1}'$ such that $\mathit{Pre}(\widetilde{v_1}, \widetilde{v_2})$ is true, $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$, and $\widetilde{v_2} \rightsquigarrow_2 \bot$. Let π_1 be a finite sequence such that (a1) $\widetilde{v_1} = \pi_1[1]$, (b1) $F_1(\pi_1[|\pi_1|])$ is true, and (c1) for each $1 < i \leq |\pi_1|$, $T_1(\pi_1[i-1], \pi_1[i])$ is true. Let ϖ_2 be an infinite sequence such that (a2) $\widetilde{v_2} = \varpi_2[1]$, (b2) for each $i \geq 1$, $F_2(\varpi_2[i])$ is false, and (c2) for each $i > 1$, $T_2(\varpi_2[i-1], \varpi_2[i])$ is true.

Let $c \in \mathbb{Z}$ be such that $\mathsf{fnb}(\pi_1[1], \varpi_2[1], c)$ is true. By clause (1), it must be the case that $\mathsf{inv}(0, c, \pi_1[1], \varpi_2[1])$ is true. We next show the following lemma.

Lemma 1. *Let $d \in \mathbb{Z}$, $1 \leq i \leq |\pi_1|$, and $1 \leq j$. Suppose $\mathsf{inv}(d, c, \pi_1[i], \varpi_2[j])$ and $F_1(\pi_1[i])$. Then, clauses* (6), (5), (4a), (4b), (3a), (3c) *cannot be simultaneously satisfied.*

Proof: By $F_1(\pi_1[i])$, (4a), (4b), and (5), we have that $\mathsf{sch_{TT}}(d, c, \pi_1[i], \varpi_2[j])$ or $\mathsf{sch_{FT}}(d, c, \pi_1[i], \varpi_2[j])$ must be true. In either case, by (c2), (3a), (3c) and the fact that $T_1(\pi_1[i], \pi_1[i])$, $\mathsf{inv}(d', b, \pi_1[i], \varpi_2[j+1])$ must be true for some $d' \in \mathbb{Z}$ (incidentally, $d' = d+1$ or $d' = d$). Also, by $F_1(\pi_1[i])$ and (6b), $\mathsf{wfr}(\varpi_2[j], \varpi_2[j+1])$ is true.

Repeating the above argument with j updated to $j+1$ and d updated to d', we derive that $\mathsf{wfr}(\varpi_1[j'], \varpi_2[j'+1])$ must be true for all $j' \geq j$. However, this violates the condition that wfr is a well-founded relation. Therefore, the clauses cannot be satisfied. ■

We now return to the proof of the theorem. Let $d = 0$, $a_1 = 1$ and $a_2 = 1$. Note that $\mathsf{inv}(d, c, \pi_1[a_1], \varpi_2[a_2])$ is true and $|d| \leq c$.

If $F_1(\pi_1[a_1])$ is true, then by Lemma 1, the constraint is unsatisfiable. So, suppose that $\neg F_1(\pi_1[a_1])$. By clause (5), it must be the case that either
$(\mathsf{s_{TT}})$ $\mathsf{sch_{TT}}(d, c, \pi_1[a_1], \varpi_2[a_2])$ is true;
$(\mathsf{s_{TF}})$ $\mathsf{sch_{TF}}(d, c, \pi_1[a_1], \varpi_2[a_2])$ is true; or
$(\mathsf{s_{FT}})$ $\mathsf{sch_{FT}}(d, c, \pi_1[a_1], \varpi_2[a_2])$ is true.
If $(\mathsf{s_{TT}})$ then let $d' = d$, $a_1' = a_1 + 1$, and $a_2' = a_2 + 1$. If $(\mathsf{s_{FT}})$ then let $d' = d-1$, $a_1' = a_1$, and $a_2' = a_2+1$. If $(\mathsf{s_{TF}}$ then let $d' = d+1$, $a_1' = a_1$, and $a_2' = a_2+1$. In any case, by (b2), (3a), (3b), and (3c), $\mathsf{inv}(d', c, \pi_1[a_1'], \varpi_2[a_2'])$ must be true. But by (b2) and (2), it must be the case that $|d'| \leq c$.

Repeating the argument in the above paragraph by setting $d = d'$, $a_1 = a_1'$, and $a_2 = a_2'$, we will eventually reach $a_1 = |\pi_1|$ such that $\mathsf{inv}(d, c, \pi_1[a_1], \varpi_2[a_2])$ is true for some d and a_2, because $\mathsf{sch_{FT}}$ can only become true finitely often due to the difference bound. At this point, $F_1(\pi_1[a_1])$ is true. Therefore by Lemma 1, the constraint is unsatisfiable.

■

I Proof of Theorem 3

Proof:

(if)

Suppose that ρ is a solution to $\mathcal{C}_{TIGNI}$. Let P_2' be a program whose transition relation T_2' is defined as follows:

$$T_2' = \{(\widetilde{v}, \widetilde{v_2}, \widetilde{v}, \widetilde{v_2}') \mid \exists r. \rho(\mathsf{fnr})(\widetilde{v}, \widetilde{v_2}, r) \wedge U_2(r, \widetilde{v_2}, \widetilde{v_2}')\}$$

where $|\widetilde{v}| = |\widetilde{V_1}|$ and fnr is the functional predicate variable added in (m6) of Def. 7. That is, P_2' is P_2 augmented to (1) take prophecy values as input and propagate them across the transitions, and (2) determinize the transitions by the assignment to fnr given in ρ. We write $\rightsquigarrow_2'$ for the reachability relation of P_2'.[9]

[9] The encoding given in Sec. 4.3 modified P_1 to propagate the prophecy values. Here, we modify P_2 for the job. The resulting constraint set is equivalent, but the proof becomes somewhat simpler.

Let $\mathcal{C}'_{\text{TIGNI}}$ be $\mathcal{C}_{\text{TIGNI}}$ but modified so that its occurrences of $\mathsf{fnr}(\widetilde{p}, \widetilde{V_2}, r)$ are replaced by $\rho(\mathsf{fnr})(\widetilde{p}, \widetilde{V_2}, r)$. Clearly, ρ is a solution to $\mathcal{C}'_{\text{TIGNI}}$. Also, $\mathcal{C}'_{\text{TIGNI}}$ is a k-safety constraint set (cf. Def. 3) for P_1 and P_2' against the pre-condition Pre' and the post-condition $Post'$. Therefore, by Theorem 1, P_1 and P_2' satisfies the k-safety property given by Pre' and $Post'$.

Now, suppose that $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$. Let $\widetilde{v}$ be valuations of prophecy variables (*i.e.*, $|\widetilde{v}| = |\widetilde{v_1}|$). We have $Pre'(\widetilde{v}, \widetilde{v_1}, \widetilde{v_2})$. Also, by k-safety of P_1 and P_2', either (a) $(\widetilde{v}, \widetilde{v_2}) \rightsquigarrow'_2 \bot$ or (b) there exists $\widetilde{v_2}'$ such that $(\widetilde{v}, \widetilde{v_2}) \rightsquigarrow'_2 (\widetilde{v}, \widetilde{v_2}')$ and $Post'(\widetilde{v}, \widetilde{v_1}', \widetilde{v_2}')$.

If the former is true for some $\widetilde{v}$, then we have $\widetilde{v_2} \rightsquigarrow_2 \bot$ by letting P_2 resolve the non-deterministic choices according to the execution $(\widetilde{v}, \widetilde{v_2}) \rightsquigarrow'_2 \bot$. Otherwise, (b) holds for all $\widetilde{v}$. Therefore, there exists $\widetilde{v_2}'$ such that $(\widetilde{v_1}, \widetilde{v_2}) \rightsquigarrow'_2 (\widetilde{v_1}, \widetilde{v_2}')$ and $Post'(\widetilde{v_1}, \widetilde{v_1}', \widetilde{v_2}')$. Therefore, by letting P_2 resolve the non-deterministic choices according to the execution $(\widetilde{v_1}, \widetilde{v_2}) \rightsquigarrow'_2 (\widetilde{v_1}, \widetilde{v_2}')$, we have $\widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}'$ and $Post(\widetilde{v_1}', \widetilde{v_2}')$. Therefore, P_1 and P_2 satisfies TI-GNI given by Pre and $Post$.

(only-if)

Suppose that P_1 and P_2 satisfy TI-GNI given by Pre and $Post$. Let us write $\pi : \widetilde{v} \rightsquigarrow_i \widetilde{v}'$ if π is a finite sequence witnessing the reachability relation $\widetilde{v} \rightsquigarrow_i \widetilde{v}'$. Likewise, let us write $\varpi : \widetilde{v} \rightsquigarrow_i \bot$ if ϖ is an infinite sequence witnessing the non-termination $\widetilde{v} \rightsquigarrow_i \bot$.

For each state $\widetilde{v_1}'$ of P_1, we define $L(\widetilde{v_1}')$ to be a (possibly infinite) list of finite and infinite sequences obtained by totally ordering the elements of the following set:

$$\{\pi \mid \exists \widetilde{v_1}, \widetilde{v_2}, \widetilde{v_1}'\widetilde{v_2}'. Pre(\widetilde{v_1}, \widetilde{v_2}) \wedge \widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}' \wedge \pi : \widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}' \wedge Post(\widetilde{v_1}', \widetilde{v_2}')\} \cup$$
$$\{\varpi \mid \exists \widetilde{v_1}, \widetilde{v_2}, \widetilde{v_1}'. Pre(\widetilde{v_1}, \widetilde{v_2}) \wedge \widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}' \wedge \varpi : \widetilde{v_2} \rightsquigarrow_2 \bot\}$$

That is, $L(\widetilde{v_1}')$ is a list of the terminating and non-terminating execution traces of P_2 that can match a (terminating) execution trace of P_1 whose final state is $\widetilde{v_1}'$. Note that, if $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $v_1 \rightsquigarrow_1 \widetilde{v_1}'$, then there exists $\xi \in L(\widetilde{v_1}')$ such that $\xi[1] = \widetilde{v_2}$.

For $\xi \in L(\widetilde{v_1}')$ and $1 \leq i \leq |\xi|$ (where $|\varpi| = \infty$ for an infinite trace ϖ), let $choice(\xi, i)$ be the angelic non-deterministic choice made in the i-th step of the execution ξ. For a function f, we write $f[a \mapsto b]$ for the function defined by $f[a \mapsto b](a) = b$ and $f[a \mapsto b](c) = f(c)$ for all $c \neq a$.

Next, we define a function det by the following process. Initialize $det \leftarrow \emptyset$. Then, for each state $\widetilde{v_1}'$ of P_1, apply the steps below until $L(\widetilde{v_1}')$ is empty:

1. Take the head element $\xi \in L(\widetilde{v_1}')$.
2. Scan ξ forwards and, at each position i, record its angelic choice by updating $det \leftarrow det[(\widetilde{v_1}', \xi[i]) \mapsto choice(\xi, i)]$ if $(\widetilde{v_1}', \xi[i]) \notin \mathrm{dom}(det)$ and otherwise leave det unchanged.

Finally, for each pair $(\widetilde{v_1}, \widetilde{v_2})$ of states of P_1 and and P_2 such that $(\widetilde{v_1}, \widetilde{v_2}) \notin \mathrm{dom}(det)$, update det by setting $det \leftarrow det[(\widetilde{v_1}, \widetilde{v_2}) \mapsto r]$ where r is arbitrary. Note that this is an infinite "process" in general since the number of states of

P_1, $|L(\widetilde{v_1}')|$, and the length of a sequence in $L(\widetilde{v_1}')$, can all be infinite. However, it is well-defined. Importantly, *det* thus constructed is a total function from the pairs of P_1 and P_2 states.

Note that using *det* as the determinizing choice function in P_2' would make P_1 and P_2' (cf. the soundness proof above) satisfy the k-safety property given by Pre' and $Post'$. This follows from the fact that, if $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$, then any execution of P_2' from $(\widetilde{v_1}', \widetilde{v_2})$ only visits states $(\widetilde{v_1}', \widetilde{v})$ where $\widetilde{v}$ occurs in $L(\widetilde{v_1}')$ and therefore may only reach an output $(\widetilde{v_1}', \widetilde{v_2}')$ such that $Post(\widetilde{v_1}', \widetilde{v_2}')$ is true.[10]

Therefore, the rest of the proof follows the structure of the completeness direction of the proof of Theorem 1. Let us call a pair of states $\widetilde{v_1}$ and $(\widetilde{v_1}', \widetilde{v_2})$ of P_1 and P_2' *initial* if $Pre'(\widetilde{v_1}', \widetilde{v_1}, \widetilde{v_2})$ is true (*i.e.*, $Pre(\widetilde{v_1}, \widetilde{v_2})$ is true). Also, in what follows, we assume that P_2' uses *det* as the determinizing choice function.

Let $\mathsf{sch_{TT}} = \mathtt{true}$ and $\mathsf{sch_{FT}} = \mathsf{sch_{TF}} = \mathtt{false}$, *i.e.*, let the scheduler be lock-step. Let fnr be the set of tuples $(\widetilde{v_1}', \widetilde{v_2}, r)$ such that $det(\widetilde{v_1}', \widetilde{v_2}) = r$, *i.e.*, fnr expresses the graph of *det*. Note that fnr trivially satisfies the function-ness requirement. Let inv be the set of tuples $(\widetilde{v_1}', \widetilde{v_1}, \widetilde{v_2})$ where $\widetilde{v_1}$ and $(\widetilde{v_1}', \widetilde{v_2})$ are pair of states of P_1 and P_2' reachable from some initial pair of states by lock-step evaluation. It is easy to see that these predicates satisfy $\mathcal{C}_{\text{TIGNI}}$.

■

J Proof of Theorem 4

Proof:

(if)

The proof is similar to the soundness direction of Theorem 3 and proceeds by determinizing P_2 using the solution to the constraint set. So, suppose that ρ is a solution to $\mathcal{C}_{TSGNI}$. Let P_2' be a program whose transition relation T_2' is defined as follows:

$$T_2' = \{(\widetilde{v}, \widetilde{v_2}, \widetilde{v}, \widetilde{v_2}') \mid \exists r. \rho(\mathsf{fnr})(\widetilde{v}, \widetilde{v_2}, r) \wedge U_2(r, \widetilde{v_2}, \widetilde{v_2}')\}$$

where $|\widetilde{v}| = |\widetilde{V_1}|$ and fnr is the functional predicate variable added in (m6) of Def. 7. That is, P_2' is P_2 augmented to (1) take prophecy values as input and propagate them across the transitions, and (2) determinize the transitions by the assignment to fnr given in ρ. We write $\rightsquigarrow_2'$ for the reachability relation of P_2'. We also consider modification of P_1, P_1', that is defined by the transition relation T_1' and the final state predicate F_1' given in Def. 8. We write $\rightsquigarrow_1'$ for

[10] However, the behavior of P_2' is not necessarily equivalent to what the traces in $L(\widetilde{v_1}')$ stipulate. For instance, P_2' may non-terminate even when all traces in $L(\widetilde{v_1}')$ are finite.

the reachability relation of P_1'. Note that, propagating the prophecy values in both P_1' and P_2' is equivalent to propagating them in one of the two, since neither transitions modify the prophecy values.

Let $\mathcal{C}'_{\mathrm{TSGNI}}$ be $\mathcal{C}_{\mathrm{TSGNI}}$ but modified so that its occurrences of $\mathsf{fnr}(\widetilde{p}, \widetilde{V_2}, r)$ are replaced by $\rho(\mathsf{fnr})(\widetilde{p}, \widetilde{V_2}, r)$. Clearly, ρ is a solution to $\mathcal{C}'_{\mathrm{TSGNI}}$. Note that $\mathcal{C}'_{\mathrm{TSGNI}}$ asserts a conjunction of the k-safety for P_1' and P_2' against the pre-condition Pre' and the post-condition $Post'$ and the co-termination for P_1' and P_2' against the pre-condition Pre'. Therefore, by Theorem 1, (a) P_1' and P_2' satisfies the k-safety property given by Pre' and $Post'$, and by Theorem 2, (b) P_1' and P_2' satisfies the co-termination property given by Pre'.

Now, suppose that $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$. We have $Pre'(\widetilde{v_1}, \widetilde{v_1}, \widetilde{v_2})$ and $(\widetilde{v_1}, \widetilde{v_1}) \rightsquigarrow'_1 (\widetilde{v_1}, \widetilde{v_1}')$. Therefore, by (b), we have that $(\widetilde{v_1}, \widetilde{v_2}) \not\rightsquigarrow'_2 \bot$. Therefore, $(\widetilde{v_1}, \widetilde{v_2}) \rightsquigarrow'_2 (\widetilde{v_1}, \widetilde{v_2}')$ for some $\widetilde{v_2}'$, assuming that every pre-related state has at least one execution. Then, by (a), it must be the case that $Post'(\widetilde{v_1}, \widetilde{v_1}', \widetilde{v_2}')$. Therefore, by letting P_2 resolve the non-deterministic choices according to the execution $(\widetilde{v_1}, \widetilde{v_2}) \rightsquigarrow'_2 (\widetilde{v_1}, \widetilde{v_2}')$, we have $\widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}'$ and $Post(\widetilde{v_1}', \widetilde{v_2}')$. Therefore, P_1 and P_2 satisfies TS-GNI given by Pre and $Post$.

(only-if)

The proof is similar to the completeness direction of Theorem 3 and proceeds by constructing a determinizing choice function F from the executions of the two programs. We will use some of the notations defined there. That is, we write $\pi : \widetilde{v} \rightsquigarrow_i \widetilde{v}'$ if π is a finite sequence witnessing the reachability relation $\widetilde{v} \rightsquigarrow_i \widetilde{v}'$, and when π is an execution of P_2 and $1 \leq i \leq |\pi|$, we write $\mathit{choice}(\pi, i)$ be the angelic non-deterministic choice made in the i-th step of π. Also, for a function f, we write $f[a \mapsto b]$ for the function defined by $f[a \mapsto b](a) = b$ and $f[a \mapsto b](c) = f(c)$ for all $c \neq a$.

Now we proceed with the proof of the completeness direction. So, suppose that P_1 and P_2 satisfy TI-GNI given by Pre and $Post$. For each state $\widetilde{v_1}'$ of P_1, we define $L(\widetilde{v_1}')$ to be a (possibly infinite) list of *finite* sequences obtained by totally ordering the elements of the following set:

$$\{\pi \mid \exists \widetilde{v_1}, \widetilde{v_2}, \widetilde{v_1}'\widetilde{v_2}'. Pre(\widetilde{v_1}, \widetilde{v_2}) \wedge \widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}' \wedge \pi : \widetilde{v_2} \rightsquigarrow_2 \widetilde{v_2}' \wedge Post(\widetilde{v_1}', \widetilde{v_2}')\}$$

That is, $L(\widetilde{v_1}')$ is a list of the terminating execution traces of P_2 that can match a (terminating) execution trace of P_1 whose final state is $\widetilde{v_1}'$. Note that, if $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $v_1 \rightsquigarrow_1 \widetilde{v_1}'$, then there exists $\pi \in L(\widetilde{v_1}')$ such that $\pi[1] = \widetilde{v_2}$.

Next, we define a function det by the following process. Initialize $\mathit{det} \leftarrow \emptyset$. Then, for each state $\widetilde{v_1}'$ of P_1, apply the steps below until $L(\widetilde{v_1}')$ is empty:

1. Take the head element $\pi \in L(\widetilde{v_1}')$.
2. Scan π forwards and, at each position i, record its angelic choice by updating $\mathit{det} \leftarrow \mathit{det}[(\widetilde{v_1}', \pi[i]) \mapsto \mathit{choice}(\pi, i)]$.

Finally, for each pair $(\widetilde{v_1}, \widetilde{v_2})$ of states of P_1 and and P_2 such that $(\widetilde{v_1}, \widetilde{v_2}) \notin \mathrm{dom}(\mathit{det})$, update det by setting $\mathit{det} \leftarrow \mathit{det}[(\widetilde{v_1}, \widetilde{v_2}) \mapsto r]$ where r is arbitrary. As that in the proof of Theorem 3, this is an infinite "process" in general

since the number of states of P_1 and $|L(\widetilde{v_1}')|$ can both be infinite. However, it is well-defined. Importantly, *det* thus constructed is a total function from the pairs of P_1 and P_2 states.

An important difference from the construction of *det* given in Theorem 3 is that in step 2., we always update *det* by the angelic choice, *i.e.*, even if $(\widetilde{v_1}', \xi[i]) \in \mathrm{dom}(det)$. This ensures the "co-termination" of P_2' when it is determinized by *det*.[11]

Note that using *det* as the determinizing choice function in P_2' would make P_1' and P_2' (cf. the soundness proof above) satisfy the k-safety property given by Pre' and $Post'$ and the co-termination property given by Pre'. This follows from the fact that, if $Pre(\widetilde{v_1}, \widetilde{v_2})$ and $\widetilde{v_1} \rightsquigarrow_1 \widetilde{v_1}'$, then any execution of P_2' from $(\widetilde{v_1}', \widetilde{v_2})$ only visits states $(\widetilde{v_1}', \widetilde{v})$ where $\widetilde{v}$ occurs in $L(\widetilde{v_1}')$ and reaches an output $(\widetilde{v_1}', \widetilde{v_2}')$ such that $Post(\widetilde{v_1}', \widetilde{v_2}')$ is true. In particular, the termination of P_2' from such a state $(\widetilde{v_1}', \widetilde{v_2})$ is guaranteed by the fact that all traces in $L(\widetilde{v_1}')$ are finite.

Therefore, the rest of the proof follows the structures of the completeness directions of the proofs of Theorems 1 and 2. Let us call a pair of states $(\widetilde{v_1}', \widetilde{v_1})$ and $(\widetilde{v_1}', \widetilde{v_2})$ of P_1' and P_2' *initial* if $Pre'(\widetilde{v_1}', \widetilde{v_1}, \widetilde{v_2})$ is true (*i.e.*, $Pre(\widetilde{v_1}, \widetilde{v_2})$ is true). Also, in what follows, we assume that P_2' uses *det* as the determinizing choice function.

Let $\mathsf{sch}_{\mathsf{TT}} = \mathtt{true}$ and $\mathsf{sch}_{\mathsf{FT}} = \mathsf{sch}_{\mathsf{TF}} = \mathtt{false}$, *i.e.*, let the scheduler be lock-step. Let fnr be the set of tuples $(\widetilde{v_1}', \widetilde{v_2}, r)$ such that $F(\widetilde{v_1}', \widetilde{v_2}) = r$, *i.e.*, fnr expresses the graph of F. Note that fnr trivially satisfies the functionness requirement. Let $\mathsf{fnb}(\widetilde{V}, b) = b = 0$ (any other predicate that sets b to be non-negative also works). Let inv be the set of tuples $(d, 0, \widetilde{v_1}', \widetilde{v_1}, \widetilde{v_2})$ where $\widetilde{v_1}', \widetilde{v_1}$ and $(\widetilde{v_1}', \widetilde{v_2})$ are reachable from some initial pair of states by lock-step evaluation, and $d = 0$ if $\neg F_1'(\widetilde{v_1}', \widetilde{v_1}) \wedge \neg F_2'(\widetilde{v_1}', \widetilde{v_2})$ (d is arbitrary if $F_1'(\widetilde{v_1}', \widetilde{v_1}) \vee F_2'(\widetilde{v_1}', \widetilde{v_2})$). Let R be the set of states of P_2' reachable from some initial pair of states after the corresponding execution of P_1' has terminated by lock-step evaluation. That is, R is the set of states $(\widetilde{v_1}', \widetilde{v_2})$ satisfying the following: there exist a state $(\widetilde{v_1}', \widetilde{v_1})$ such that $((\widetilde{v_1}', \widetilde{v_1}), (\widetilde{v_1}', \widetilde{v_2}))$ is reachable from some initial pair of states by lock-step evaluation and $F_1'(\widetilde{v_1}', \widetilde{v_1})$ is true. Let $\mathsf{wfr} = (R \times R) \cap \{((\widetilde{v_1}', \widetilde{v_2}), (\widetilde{v_1}', \widetilde{v_2}')) \mid T_2'((\widetilde{v_1}', \widetilde{v_2}), (\widetilde{v_1}', \widetilde{v_2}'))\}$ (or any other well-founded relation witnessing the termination of R). It is easy to see that these predicates satisfy $\mathcal{C}_{\mathrm{TSGNI}}$.

■

K Unsatisfiability Checking of Example Instances

The unsatisfiability of the given example instances $(\mathcal{E}, \mathcal{K})$ can be decided by an off-the-shelf SAT solver if $\mathcal{E}$ has only ordinary predicate variables because $\mathcal{E}$ is a finite set of clauses not containing term variables. Otherwise, we use the following

[11] The above "overwriting" construction of *det* actually also works for the proof of Theorem 3. However, the construction method given there does not work here because it may introduce unwanted non-termination.

(CDCL-like) iterative algorithm staring from $\mathcal{E}_0 = \mathcal{E}$: For each iteration $i \geq 0$, we first check whether $(\mathcal{E}_i, \emptyset)$ is unsatisfiable. If so, then we conclude that $(\mathcal{E}, \mathcal{K})$ is unsatisfiable. Otherwise, we obtain a satisfying assignment σ for $\mathcal{E}_i$. Then, for each well-founded predicate variable X, we consider the graph comprising the edges $\{\, (\widetilde{v}_1, \widetilde{v}_2) \mid \models \sigma(X(\widetilde{v}_1, \widetilde{v}_2)) \,\}$ and enumerate its simple cycles (e.g., by using the algorithm of [35]). Note that such cycles would be counterexamples to the well-foundedness constraint X. Also, for each functional predicate variable X, we enumerate the pairs $\{X(\widetilde{v}, v_1), X(\widetilde{v}, v_2)\}$ such that $v_1 \neq v_2$ and $\models \sigma(X(\widetilde{v}, v_1) \wedge X(\widetilde{v}, v_2))$, which would be counterexamples to the functionality constraint X. If no such cycles nor pairs exist, we conclude that $(\mathcal{E}, \mathcal{K})$ is satisfiable. Otherwise, we let $\mathcal{E}_{i+1}$ be $\mathcal{E}_i$ but with the following new learnt clauses added:

- $\neg X(\widetilde{v}_1, \widetilde{v}_2) \vee \cdots \vee \neg X(\widetilde{v}_{m-1}, \widetilde{v}_m)$ for each simple cycle $\widetilde{v}_1, \ldots, \widetilde{v}_m = \widetilde{v}_1$ of each well-founded predicate X, and
- $\neg X(\widetilde{v}, v_1) \vee \neg X(\widetilde{v}, v_2)$ for each pair $\{X(\widetilde{v}, v_1), X(\widetilde{v}, v_2)\}$ of each functional predicate X.

We then proceed to the next iteration with $\mathcal{E}_{i+1}$.

It is worth mentioning here that if the original pfwCSP $(\mathcal{C}, \mathcal{K})$ is unsatisfiable and $\mathcal{C}$ has no well-founded predicate variable, there always exists an unsatisfiable finite set $\mathcal{E}$ of example instances of $\mathcal{C}$. However, there is, in general, no such finite witness of the unsatifiability if $\mathcal{C}$ has a well-founded predicate variable.

L A Refined Stratified Template Family for Well-Founded Predicates

The stratified template family $T_X^{\Downarrow}$ for well-founded predicates shown in Fig. 2 can be further refined without loss of generality by simplifying and using $\mathsf{if}\ r(\widetilde{x}) \geq 0\ \mathsf{then}\ r(\widetilde{x})\ \mathsf{else}\ -1$ instead of $r(\widetilde{x})$.

$$
\begin{aligned}
T_X^{\Downarrow}(np, nl, nc, rc, rd, dc, dd) \triangleq \lambda(\widetilde{x}, \widetilde{y}).\ & \left(\bigvee_{j=1}^{np} D_j(\widetilde{y})\right) \\
& \wedge \left(\bigvee_{i=1}^{np} D_i(\widetilde{x}) \wedge \bigwedge_{j=1}^{np} (D_j(\widetilde{y}) \Rightarrow DEC_{i,j}(\widetilde{x}, \widetilde{y}))\right) \\
DEC_{i,j}(\widetilde{x}, \widetilde{y}) \triangleq \bigvee_{k=1}^{nl} & \begin{pmatrix} r_{i,k}(\widetilde{x}) \geq 0 \wedge r_{i,k}(\widetilde{x}) > r_{j,k}(\widetilde{y}) \wedge \\ \bigwedge_{\ell=1}^{k-1} (r_{i,\ell}(\widetilde{x}) < 0 \wedge r_{j,\ell}(\widetilde{y}) < 0 \vee r_{i,\ell}(\widetilde{x}) \geq r_{j,\ell}(\widetilde{y})) \end{pmatrix}
\end{aligned}
$$

M Relational Verification Benchmarks

Our relational verification benchmark set consists of:

- The k-safety verification problem `DoubleSquareNI_h**` from Example 1, which is originally introduced in [51].
- The k-safety verification problem `HalfSquareNI` of the following program obtained from [51]:

```
pre(low1 == low2)
halfSquare(int h, int low) {
  assume(low > h > 0);
  int i = 0, y = 0, v = 0;
  while (h > i) {
    i++; y += y;
  }
  v = 1;
  while (low > i) {
    i++; y += y;
  }
  return y;
}
post(y1 == y2)
```

The encoded constraints are:

```
Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2) :-
  low1 = low2, low1 > h1, h1 > 0, low2 > h2, h2 > 0,
  b1, i1 = 0, y1 = 0, v1 = 0,
  b2, i2 = 0, y2 = 0, v2 = 0.

Inv(b1' : bool, h1, low1, i1', y1', v1', b2 : bool, h2, low2, i2, y2, v2) :-
  Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  SchTF(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  b1 and h1 > i1 and b1' and i1' = i1 + 1 and y1' = y1 + y1 and v1' = v1 or
  b1 and h1 <= i1 and !b1' and i1' = i1 and y1' = y1 and v1' = 1 or
  !b1 and low1 > i1 and !b1' and i1' = i1 + 1 and y1' = y1 + y1 and v1' = v1 or
  !b1 and low1 <= i1 and !b1' and i1' = i1 and y1' = y1 and v1' = v1.
Inv(b1 : bool, h1, low1, i1, y1, v1, b2' : bool, h2, low2, i2', y2', v2') :-
  Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  SchFT(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  b2 and h2 > i2 and b2' and i2' = i2 + 1 and y2' = y2 + y2 and v2' = v2 or
  b2 and h2 <= i2 and !b2' and i2' = i2 and y2' = y2 and v2' = 1 or
  !b2 and low2 > i2 and !b2' and i2' = i2 + 1 and y2' = y2 + y2 and v2' = v2 or
  !b2 and low2 <= i2 and !b2' and i2' = i2 and y2' = y2 and v2' = v2.
Inv(b1' : bool, h1, low1, i1', y1', v1', b2' : bool, h2, low2, i2', y2', v2') :-
  Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  SchTT(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  b1 and h1 > i1 and b1' and i1' = i1 + 1 and y1' = y1 + y1 and v1' = v1 or
  b1 and h1 <= i1 and !b1' and i1' = i1 and y1' = y1 and v1' = 1 or
  !b1 and low1 > i1 and !b1' and i1' = i1 + 1 and y1' = y1 + y1 and v1' = v1 or
  !b1 and low1 <= i1 and !b1' and i1' = i1 and y1' = y1 and v1' = v1,
  b2 and h2 > i2 and b2' and i2' = i2 + 1 and y2' = y2 + y2 and v2' = v2 or
  b2 and h2 <= i2 and !b2' and i2' = i2 and y2' = y2 and v2' = 1 or
  !b2 and low2 > i2 and !b2' and i2' = i2 + 1 and y2' = y2 + y2 and v2' = v2 or
```

```
    !b2 and low2 <= i2 and !b2' and i2' = i2 and y2' = y2 and v2' = v2.

  b1 or low1 > i1 :-
    Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    SchTF(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    b2 or low2 > i2.
  b2 or low2 > i2 :-
    Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    SchFT(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    b1 or low1 > i1.
  SchTF(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  SchFT(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
  SchTT(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2) :-
    Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    b1 or low1 > i1 or b2 or low2 > i2.

  y1 = y2 :-
    Inv(b1 : bool, h1, low1, i1, y1, v1, b2 : bool, h2, low2, i2, y2, v2),
    !b1, low1 <= i1, !b2, low2 <= i2.
```

– The k-safety verification problem `ArrayInsert` of the following program obtained from [51] by simulating the control flow of the original array-manipulating program:

```
pre(len1 == len2)
int arrayInsert(int len, int h) {
  int i=0;
  while (i < len && i != h) i++;
  len = len + 1;
  while (i < len) i++;
  return i;
}
post(i1 == i2)
```

The encoded constraints are:

```
Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2) :-
  len1 = len2, b1, i1 = 0, b2, i2 = 0.

Inv(b1' : bool, len1', h1, i1', b2 : bool, len2, h2, i2) :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  SchTF(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b1 and i1 < len1 and i1 <> h1 and b1' and len1' = len1 and i1' = i1 + 1 or
  b1 and (i1 >= len1 or i1 = h1) and !b1' and len1' = len1 + 1 and i1' = i1 or
  !b1 and i1 < len1 and !b1' and len1' = len1 and i1' = i1 + 1 or
```

```
  !b1 and i1 >= len1 and !b1' and len1' = len1 and i1' = i1.
Inv(b1 : bool, len1, h1, i1, b2' : bool, len2', h2, i2') :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  SchFT(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b2 and i2 < len2 and i2 <> h2 and b2' and len2' = len2 and i2' = i2 + 1 or
  b2 and (i2 >= len2 or i2 = h2) and !b2' and len2' = len2 + 1 and i2' = i2 or
  !b2 and i2 < len2 and !b2' and len2' = len2 and i2' = i2 + 1 or
  !b2 and i2 >= len2 and !b2' and len2' = len2 and i2' = i2.
Inv(b1' : bool, len1', h1, i1', b2' : bool, len2', h2, i2') :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  SchTT(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b1 and i1 < len1 and i1 <> h1 and b1' and len1' = len1 and i1' = i1 + 1 or
  b1 and (i1 >= len1 or i1 = h1) and !b1' and len1' = len1 + 1 and i1' = i1 or
  !b1 and i1 < len1 and !b1' and len1' = len1 and i1' = i1 + 1 or
  !b1 and i1 >= len1 and !b1' and len1' = len1 and i1' = i1,
  b2 and i2 < len2 and i2 <> h2 and b2' and len2' = len2 and i2' = i2 + 1 or
  b2 and (i2 >= len2 or i2 = h2) and !b2' and len2' = len2 + 1 and i2' = i2 or
  !b2 and i2 < len2 and !b2' and len2' = len2 and i2' = i2 + 1 or
  !b2 and i2 >= len2 and !b2' and len2' = len2 and i2' = i2.

b1 or i1 < len1 :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  SchTF(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b2 or i2 < len2.
b2 or i2 < len2 :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  SchFT(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b1 or i1 < len1.
SchTF(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
SchFT(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
SchTT(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2) :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  b1 or i1 < len1 or b2 or i2 < len2.

i1 = i2 :-
  Inv(b1 : bool, len1, h1, i1, b2 : bool, len2, h2, i2),
  !b1, i1 >= len1, !b2, i2 >= len2.
```

– The k-safety verification problem `SquareSum` of the following program obtained from [51] by replacing the nonlinear expression `a*a` in the original program with `a`:

```
pre(a1 < a2 && b2 < b1)
squaresSum(int a, int b) {
  assume(0 < a < b);
  int c=0;
```

```
    while (a<b) { c+=a; a++; }
    return c;
  }
  post(c2 < c1)
```

The encoded constraints are:

```
Inv(a1, b1, c1, a2, b2, c2) :-
  a1 < a2, b2 < b1,
  0 < a1, a1 < b1, 0 < a2, a2 < b2,
  c1 = 0, c2 = 0.

Inv(a1', b1, c1', a2, b2, c2) :-
  Inv(a1, b1, c1, a2, b2, c2),
  SchTF(a1, b1, c1, a2, b2, c2),
  a1 < b1 and c1' = c1 + a1 and a1' = a1 + 1 or a1 >= b1 and c1' = c1 and a1' = a1.
Inv(a1, b1, c1, a2', b2, c2') :-
  Inv(a1, b1, c1, a2, b2, c2),
  SchFT(a1, b1, c1, a2, b2, c2),
  a2 < b2 and c2' = c2 + a2 and a2' = a2 + 1 or a2 >= b2 and c2' = c2 and a2' = a2.
Inv(a1', b1, c1', a2', b2, c2') :-
  Inv(a1, b1, c1, a2, b2, c2),
  SchTT(a1, b1, c1, a2, b2, c2),
  a1 < b1 and c1' = c1 + a1 and a1' = a1 + 1 or a1 >= b1 and c1' = c1 and a1' = a1,
  a2 < b2 and c2' = c2 + a2 and a2' = a2 + 1 or a2 >= b2 and c2' = c2 and a2' = a2.

a1 < b1 :-
  Inv(a1, b1, c1, a2, b2, c2),
  SchTF(a1, b1, c1, a2, b2, c2), a2 < b2.
a2 < b2 :-
  Inv(a1, b1, c1, a2, b2, c2),
  SchFT(a1, b1, c1, a2, b2, c2), a1 < b1.
SchTF(a1, b1, c1, a2, b2, c2),
SchFT(a1, b1, c1, a2, b2, c2),
SchTT(a1, b1, c1, a2, b2, c2) :-
  Inv(a1, b1, c1, a2, b2, c2),
  a1 < b1 or a2 < b2.

c2 < c1 :- Inv(a1, b1, c1, a2, b2, c2), a1 >= b1, a2 >= b2.
```

In the experiment, we provided the following constraint as a hint:

```
a1 > 0, b2 < b1 :- Inv(a1, b1, c1, a2, b2, c2).
```

– The co-termination verification problem `CotermIntro` from Example 2.

– The TS-GNI verification problem `TS_GNI_h**` from Example 3. In the experiment of `TS_GNI_hFT`, we provided the following constraint as a hint:

```
x1 >= low1 :- Inv(pr, d, b, b1 : bool, x1, low1, b2 : bool, x2, low2).
```

In the experiment of `TS_GNI_hTT`, we provided the following constraint as a hint:

```
b1 or x1 >= low1 :- Inv(pr, d, b, b1 : bool, x1, low1, b2 : bool, x2, low2).
```

Note that these are a part of necessary non-relational invariant.

– The TS-GNI verification problem `SimpleTS_GNI1` of the following program:

```
while ( * ) { x ++; }; return (high + x);
```

The encoded constraints are:

```
Inv(0, b, b1 : bool, x1, high1, b2 : bool, x2, high2) :-
  FN_DB(x1, high1, x2, high2, b), b1, b2, x1 = x2.
Inv(d', b, b1' : bool, x1', high1, b2 : bool, x2, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1 and (b1' and x1' = x1 + 1 or !b1' and x1' = x1) or
  !b1 and !b1' and x1' = x1,
  (!b1 or !b2 or d' = d + 1).
Inv(d', b, b1 : bool, x1, high1, b21 : bool, x21, high2),
Inv(d', b, b1 : bool, x1, high1, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and !b22 and x22 = x1,
  (!b1 or !b2 or d' = d - 1).
Inv(d, b, b1' : bool, x1', high1, b21 : bool, x21, high2),
Inv(d, b, b1' : bool, x1', high1, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1 and (b1' and x1' = x1 + 1 or !b1' and x1' = x1) or
  !b1 and !b1' and x1' = x1,
  b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and !b22 and x22 = x1.

b1 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b2.
```

```
b2 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1.
SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
SchTT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), b1 or b2.
-b <= d and d <= b and b >= 0 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), b1, b2.

high1 + x1 = high2 + x2 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), !b1, !b2.

WF_R2(b2 : bool, x2, high2, b21 : bool, x21, high2),
WF_R2(b2 : bool, x2, high2, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  !b1, b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2.
```

– The TS-GNI verification problem `SimpleTS_GNI2` of the following program:

```
x = high; while ( * ) { x ++; }; return x;
```

The encoded constraints are:

```
Inv(0, b, b1 : bool, x1, high1, b2 : bool, x2, high2) :-
  FN_DB(x1, high1, x2, high2, b), b1, b2, x1 = high1, x2 = high2.
Inv(d', b, b1' : bool, x1', high1, b2 : bool, x2, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1 and (b1' and x1' = x1 + 1 or !b1' and x1' = x1) or
  !b1 and !b1' and x1' = x1,
  (!b1 or !b2 or d' = d + 1).
Inv(d', b, b1 : bool, x1, high1, b21 : bool, x21, high2),
Inv(d', b, b1 : bool, x1, high1, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and !b22 and x22 = x1,
  (!b1 or !b2 or d' = d - 1).
Inv(d, b, b1' : bool, x1', high1, b21 : bool, x21, high2),
Inv(d, b, b1' : bool, x1', high1, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1 and (b1' and x1' = x1 + 1 or !b1' and x1' = x1) or
```

```
  !b1 and !b1' and x1' = x1,
  b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and !b22 and x22 = x1.

b1 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b2.
b2 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  b1.
SchTF(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
SchFT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
SchTT(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), b1 or b2.
-b <= d and d <= b and b >= 0 :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), b1, b2.

x1 = x2 :- Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2), !b1, !b2.

WF_R2(b2 : bool, x2, high2, b21 : bool, x21, high2),
WF_R2(b2 : bool, x2, high2, b22 : bool, x22, high2) :-
  Inv(d, b, b1 : bool, x1, high1, b2 : bool, x2, high2),
  !b1, b2 and b21 and x21 = x2 + 1 and !b22 and x22 = x2.
```

– The TS-GNI verification problem `InfBranchTS_GNI` of the following program:

```
if (high) {
  while (x>0) { x = x - max( * , 1); }
} else {
  while (x>0) { x = x - 1; }
}
```

The encoded constraints are:

```
Inv(0, b, high1 : bool, x1, high2 : bool, x2) :-
  FN_DB(high1 : bool, x1, high2 : bool, x2, b), x1 = x2.
Inv(d', b, high1 : bool, x1', high2 : bool, x2) :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  SchTF(d, b, high1 : bool, x1, high2 : bool, x2),
  high1 and x1 > 0 and FN_R(high1 : bool, x1, nd) and
  (nd >= 1 and x1' = x1 - nd or x1' = x1 - 1) and d' = d + 1 or
  !high1 and x1 > 0 and x1' = x1 - 1 or
  x1 <= 0 and x1' = x1,
```

```
  (x1 <= 0 or x2 <= 0 or d' = d + 1).
Inv(d', b, high1 : bool, x1, high2 : bool, x2') :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  SchFT(d, b, high1 : bool, x1, high2 : bool, x2),
  high2 and x2 > 0 and nd >= 1 and x2' = x2 - nd or
  !high2 and x2 > 0 and x2' = x2 - 1 or
  x2 <= 0 and x2' = x2,
  (x1 <= 0 or x2 <= 0 or d' = d - 1).
Inv(d, b, high1 : bool, x1', high2 : bool, x2') :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  SchTT(d, b, high1 : bool, x1, high2 : bool, x2),
  high1 and x1 > 0 and FN_R(high1 : bool, x1, nd) and
  (nd >= 1 and x1' = x1 - nd or x1' = x1 - 1) or
  !high1 and x1 > 0 and x1' = x1 - 1 or
  x1 <= 0 and x1' = x1,
  high2 and x2 > 0 and nd >= 1 and x2' = x2 - nd or
  !high2 and x2 > 0 and x2' = x2 - 1 or
  x2 <= 0 and x2' = x2.

x1 > 0 :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  SchTF(d, b, high1 : bool, x1, high2 : bool, x2),
  x2 > 0.
x2 > 0 :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  SchFT(d, b, high1 : bool, x1, high2 : bool, x2),
  x1 > 0.
SchTF(d, b, high1 : bool, x1, high2 : bool, x2),
SchFT(d, b, high1 : bool, x1, high2 : bool, x2),
SchTT(d, b, high1 : bool, x1, high2 : bool, x2) :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2), x1 > 0 or x2 > 0.
-b <= d and d <= b and b >= 0 :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2), x1 > 0, x2 > 0.

top :- Inv(d, b, high1 : bool, x1, high2 : bool, x2), x1 <= 0, x2 <= 0.

WF_R1(high1 : bool, x1, high1 : bool, x1') :-
  Inv(d, b, high1 : bool, x1, high2 : bool, x2),
  x2 <= 0,
  high1 and x1 > 0 and FN_R(high1 : bool, x1, nd) and
  (nd >= 1 and x1' = x1 - nd or x1' = x1 - 1) or
  !high1 and x1 > 0 and x1' = x1 - 1.
```

– The TI-GNI verification problem `TI_GNI_h**` of the following program:

```
if (high) {
```

```
    x = *;
    if (x >= low) { return x; } else { return low; }
  } else {
    x = low;
    while ( * ) { x++; }
    return x;
  }
```

The encoded constraints are:

```
Inv(pr (* prophecy variable for the return value of Copy 1 *),
    b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2) :-
  b1, b2, low1 = low2,
  high1 and x1 = nd1 or
  !high1 and x1 = low1,
  high2 and FN_R(pr, high2 : bool, low2, x2) or
  !high2 and x2 = low2.
Inv(pr, b1' : bool, x1', high1 : bool, low1, b2 : bool, x2, high2 : bool, low2) :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  SchTF(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  high1 and b1 and (x1 >= low1 and !b1' and x1' = x1 or
                    x1 < low1 and !b1' and x1' = low1) or
  !high1 and b1 and (b1' and x1' = x1 + 1 or
                     !b1' and x1' = x1) or
  !b1 and !b1' and x1' = x1.
Inv(pr, b1 : bool, x1, high1 : bool, low1, b21 : bool, x21, high2 : bool, low2),
Inv(pr, b1 : bool, x1, high1 : bool, low1, b22 : bool, x22, high2 : bool, low2) :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  SchFT(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  high2 and b2 and (x2 >= low2 and !b21 and x21 = x2 and
                                  !b22 and x22 = x2 or
                    x2 < low2 and !b21 and x21 = low2 and
                                 !b22 and x22 = low2) or
  !high2 and b2 and b21 and x21 = x2 + 1 and
                    !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and
          !b22 and x22 = x2.
Inv(pr, b1' : bool, x1', high1 : bool, low1, b21 : bool, x21, high2 : bool, low2),
Inv(pr, b1' : bool, x1', high1 : bool, low1, b22 : bool, x22, high2 : bool, low2) :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  SchTT(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  high1 and b1 and (x1 >= low1 and !b1' and x1' = x1 or
                    x1 < low1 and !b1' and x1' = low1) or
  !high1 and b1 and (b1' and x1' = x1 + 1 or
                     !b1' and x1' = x1) or
  !b1 and !b1' and x1' = x1,
```

```
  high2 and b2 and (x2 >= low2 and !b21 and x21 = x2 and
                                    !b22 and x22 = x2 or
                    x2 < low2 and !b21 and x21 = low2 and
                                  !b22 and x22 = low2) or
  !high2 and b2 and b21 and x21 = x2 + 1 and
                    !b22 and x22 = x2 or
  !b2 and !b21 and x21 = x2 and
          !b22 and x22 = x2.

b1 or pr <> x1 :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  SchTF(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  b2.
b2 :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  SchFT(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  b1 or pr <> x1.
SchTF(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
SchFT(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
SchTT(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2) :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  b1 or pr <> x1 or b2.

x1 = x2 :-
  Inv(pr, b1 : bool, x1, high1 : bool, low1, b2 : bool, x2, high2 : bool, low2),
  !b1 and pr = x1 (* if the prophecy is correct *), !b2.
```

In the experiment of `TI_GNI_hFT`, we provided the following constraint as a hint:

```
  x1 >= low1 :- Inv(pr, b1 : bool, x1, low1, b2 : bool, x2, low2).
```

Note that this is a part of necessary non-relational invariant.